Formal Modelling and Analysis of the NORM Multicast Protocol Using Real-Time Maude

Elisabeth Lien

Department of Linguistics
University of Oslo

May 29, 2009

Contents

Chapter 1

Introduction

Over the past decades, society has become increasingly dependent on computer technology. In our everyday life, whether taking the underground to work, buying a plane ticket over the Internet, or turning on the heat in our homes, we are in direct or indirect contact with computer systems. The increasing dependency means that it will become more and more important to ensure that computer systems function correctly. At the same time, computer systems are becoming more and more complex, which makes it very hard to control their behaviour. In particular, most larger computer systems are *distributed* systems, which are notoriously hard to understand and to program. During the design process, it is difficult to foresee every consequence of the design choices being made. Extensive testing and simulation of a system can uncover many faults, but it is often not enough to guarantee that the system is free of errors. To complement other techniques for constructing reliable systems, developers can use *formal methods*—mathematically-based techniques for specification and verification. Many formal methods provide powerful and very high level *specification languages*. By formally specifying a computer system, one can analyse the *specification* early in the system development process, and not just the *implementation* at the end of the process. The formalisation of a specification can reveal ambiguities and inconsistencies in the systems design. It is well known that the cost of correcting an error increases dramatically the later in the development process the error is discovered.

This report is a slightly modified version of the thesis [**?**], and gives an example of the use of formal methods in modelling and analysing a communication protocol under development. More specifically, I will use the executable specification language *Real-Time Maude* [1, 27] to formally specify the *NORM reliable multicast protocol* [7, 6], and to analyse the resulting specification with the Real-Time Maude analysis tool.

1.1 Task and Goals

The process of analysing the NORM multicast protocol consists of two parts:

1. Turning the informal specification of the NORM protocol into a formal specification in

the Real-Time Maude language, and

2. Analysing the resulting prototype of the protocol by means of the simulation, search and model checking tools of the Real-Time Maude system.

By using a formal method to formally specify and analyse the NORM protocol, I wish to

- gain an understanding of the protocol, make implicit knowledge in the informal specification explicit, and uncover any ambiguities or parts of the informal description that appear not to be fully specified, and
- check, by means of different analysis techniques, whether the protocol meets the requirements stated in the informal specification.

In addition to demonstrating the use of a formal method in modelling and analysing a communication protocol, the report will also examine the applicability of the Real-Time Maude specification language and tool for formal specification and analysis of such protocols.

As we will see during the course of the work, there are some parts of the informal NORM specification that appear not to have been fully specified. These become apparent during the modelling, and the formal analysis of the model serves to reinforce these finds. For parts of the protocol, it turns out to be difficult to perform an analysis, because the analysis tools demand more computer capacity than I have at my disposal. The Real-Time Maude formalism is in general very intuitive and easy to use, and has powerful analysis tools. In Chapter 10, I will summarise and discuss the results of the modelling and analysis work. The starting point of the modelling and analysis work is that of a nonexpert.

1.2 Formal Methods

The field of *formal methods* is concerned with developing mathematically-based languages, techniques, and tools for specification and verification of hardware and software systems [12]. For a long time it was considered difficult, if not impossible, to use formal methods on anything but textbook problems. A lot of effort had been put into *program verification*, but the results were not satisfying and led to few applicable techniques for real-life programs. However, research in other parts of the field has produced techniques that have been successfully applied in analysing and verifying industrial-size systems, and today many manufacturers of hardware and software systems routinely use formal methods in developing their products. For example, NASA has a formal methods research group, at the NASA Langley Research Center, which develops specification and analysis techniques for the aerospace and aviation industry [11].

An important part of the research in formal methods has focused on developing more powerful and simple *specification formalisms*, to make it easier to model and analyse distributed systems. Such systems have often been specified informally, and efforts to understand and analyse them have made use of "paper-and-pencil" methods. Formal specification languages, with a mathematically defined syntax and semantics, have been developed in order to formally specify systems at a high level of abstraction. The process of formal specification leads to a deeper understanding of a system, and makes it possible to discover inconsistencies and ambiguities in the the system's design. With *executable* specification languages, the process results in a prototype of the system, which can be subjected to further analysis. By specifying the desired properties of a system as a formal *requirement specification*, informal statements of how the system should behave are made rigorous and unambiguous. Given a formal *model* ("operational specification") of the system, and a formal requirement specification stating some properties the system should satisfy, the *verification* (or *model checking*) problem is to check whether the model of the system satisfies the requirements.

A lot of work has been put into developing techniques and tools for automated and semi-automated verification. *Model checking techniques* use state exploration techniques to check if a finite-state model of a system satisfies a given property. If the property fails the model checker provides a counterexample, and model checking is thus very useful for finding errors. *Theorem proving*, on the other hand, makes it possible to verify systems with infinite state spaces. The system and the desired properties are expressed as formulas of a mathematical logic, and the theorem prover tries to find a proof for the property from the axioms of the logic. Whereas model checking is automatic, theorem provers often require human assistance in finding a proof. Theorem proving is usually considered to be a fairly hard task for nontrivial systems, although some success stories do exist. Therefore, verification is normally used only on (safety) critical parts of a system, and only *after* a range of analysis methods such as prototyping and state space exploration has been used to eliminate most errors of the system.

Model checking techniques have proved to be very successful for verification of *hardware components* and *communication protocols*. For example, model checking is useful for finding attacks on *security protocols*. The goal of such protocols is to allow a group of agents to establish each other's identity in order to communicate safely, but it is difficult to design a protocol which is reliable despite attacks from an intruder. By constructing a model of the protocol with a limit on the number of protocol participants and the number of protocol rounds, and a model of an intruding agent, one can use a model checker to explore the state space [18]. A well-known example of how model checking can be used in analysing protocols is the *Needham-Schroeder public-key authentication protocol* (NSPK). The protocol was widely recognised and used for many years. However, in the mid 1990s, Gavin Lowe found an attack on the NSPK protocol using a model checker for CSP [16, 17].

1.3 Choosing a Modelling Formalism

When I started on the thesis that is the basis for this report, I decided to model and analyse a communication protocol. I was interested in studying specification formalisms and tools that are suitable for analysing such protocols. Since *time* is an important factor in communication protocols, I looked at formalisms which have features for modelling time explicitly. I principally considered three different formalisms: CSP, Petri nets, and Real-Time Maude—all three formalisms that are suitable for modelling concurrent systems. The CSP (communicating sequential processes) formalism is a process algebra introduced by C. A. R. Hoare [31]. CSP has a machine-readable dialect for which a number of automated tools have been developed.

Petri nets, the invention of Carl Adam Petri, provides a graphical, but formal, notation for describing processes [2]. A wide range of tools exist for simulation and analysis of Petri nets. Finally, Real-Time Maude is a timed extension of the Maude modelling language and tool, which is based on the theory of rewriting logic [19].

There are a number of reasons why I chose the Real-Time Maude formalism over (the timed versions of) CSP and Petri nets, and over timed automaton-based formalisms.

First of all, Real-Time Maude seemed like a much more natural language than CSP, and easier to understand. The CSP language has built-in operators for different kinds of operations on processes, whereas Real-Time Maude gives the user complete freedom in specifying her own operations. Since I was going to model a communication protocol, where the participants use messages to interact, being able to describe asynchronous communication in a natural way was important. Modelling asynchronous communication in Real-Time Maude is easy, but seemed more difficult in CSP, whose basic communication primitives support synchronous communication. Moreover, the Real-Time Maude system is free, whereas the tools developed for CSP that I looked at are commercial.

The Petri net formalism, on the other hand, seems more intuitive than CSP, because it gives a graphical representation of a system. However, it appeared to be difficult to model large, complex systems with Petri nets. Handling a large modelling task seemed much easier in Real-Time Maude.

The Real-Time Maude specification formalism supports object-oriented specification. This is a major advantage, because it seemed convenient to model the NORM protocol as an object system, where each node is represented as an object. Another important advantage of Real-Time Maude is that both the *static parts* (the data types of the state space) and the *dynamic parts* of a system are specified in the same formalism. The data types and the dynamic behaviour of a system are described using very different formalisms in both the Petri net and the CSP framework.

Finally, finite-control timed automata are just too restrictive to model such a large and complex protocol as NORM.

This section is meant to provide some background on my choice of formalism, and I will not return to CSP and Petri nets during the course of the report. My report is first and foremost about modelling and analysing one specific communication protocol. I will make an evaluation of the Real-Time Maude language and tool based on my modelling experience, but I will not make any attempts at using this experience for comparing Real-Time Maude with other formalisms.

1.4 The NORM Protocol

The communication protocol that I am going to model in this report—NORM— is a *multicast protocol*. A communication protocol is a set of messages and rules that determine the interaction of the protocol participants. The function of a protocol is to provide a *communication service* to higher-level programs. A *multicast* service allows a program (which can be either an application program or another, higher level protocol on the same machine) to transfer data to a group of receivers connected to the network. The NORM protocol is designed to provide *reliable*, *efficient*, *robust*, and *scalable* transport of large amounts of data across IP networks. In order to provide this service, the specification of NORM defines algorithms for dealing with network errors.

The NORM protocol specification is being developed by a working group of the Internet Engineering Task Force (IETF) [3]. The group, which develops protocol standards for reliable multicast transport, has issued many draft versions of the specification. When I started work on this report, the specification draft I am using had just been released, but it is now obsolete and has been replaced by newer versions. The fact that the specification work is not finished is one of the reasons why I chose to model NORM—it would illustrate the usefulness of formal modelling and analysis as a part of the specification process. In addition, analysing a specification draft might give more interesting results than analysing a completed, thoroughly tested specification. My other reason for choosing the NORM specification is that the complex nature of multicast communication gives rise to many interesting solutions.

1.5 An Overview of the Report

The following list gives an overview of the content of each chapter of the report:

- Chapter 2 is an overview of the Real-Time Maude modelling formalism and analysis tool.
- In Chapter 3, I start by explaining what kind of network services are offered by multicast transport protocols, and how these services are implemented. I then give an overview of the service model and components of the NORM protocol.
- The first step in formally specifying the NORM protocol is to make some decisions about the structure of the model, and (since the protocol is quite large) about which components to model. We also need a model of the network and how the nodes in the network communicate. In Chapter 4, I explain the choices I made at the start of the modelling process, and present a model of the communication topology.
- Chapter 5 presents my model of the *RTT component* of the NORM protocol, which calculates a common timeout basis for the sender and receivers. I also comment on some details of the informal specification of the component.
- In Chapter 6, I use Real-Time Maude's analysis tools to simulate and formally analyse the behaviour of the RTT component. I define two different initial states for the analysis. The analysis shows that the algorithm for calculating the common timeout basis is not fully specified.
- Chapter 7 describes my model of the NORM protocol's *data and repair transmission component*. The component implements the protocol's core function: reliable transmission of data content. I also discuss some parts of the informal specification which are ambiguous or which seem not to be fully specified.

- In Chapter 8, I set out to analyse the model of the data and repair transmission, following the pattern of the analysis in Chapter 6. The high degree of nondeterminism in the protocol leads to a combinatorial explosion in the state space, which forces me to perform search and model checking within restricted time intervals. However, the rewriting analysis demonstrates the need for clarifying those parts of the informal specification which during the modelling process appeared not to be fully specified.

- In Chapter 9, I outline how the Real-Time Maude specifications of the RTT and the data transmission components can be combined by using object-oriented inheritance techniques, and by redefining some of the rewrite rules of the specifications. However, carrying out all the necessary changes in the rules is beyond the scope of this report.

- Chapter 10 summarises the results of modelling and analysing the NORM protocol, and discusses the usefulness of applying formal specification and analysis techniques to a protocol under development. I also discuss the applicability of the Real-Time Maude formalism and tool for specification and analysis of communication protocols.

1.6 Related Work

Real-Time Maude has been used to specify and analyse several communication protocols. The most notable case study is the specification and analysis of the AER/NCA protocol suite for reliable multicast in active networks [?]. The main difference between the NORM protocol and the AER/NCA protocol is that, whereas NORM only requires basic IP multicast services from the network, the AER/NCA protocol is designed to make use of “active nodes” in the multicast topology to make error recovery more efficient and scalable. The AER/NCA protocol uses *repair servers* co-located with routers inside the network, which are capable of caching data packets, and subcasting packets in response to repair requests [?]. Although the NORM protocol can also take advantage of assistance from the network, it is designed to be capable of operating without such assistance [6].

While the underlying communication infrastructure can be quite different from protocol to protocol, it would be very advantageous to have a set of useful and general specification techniques for different kinds of communication protocols. An implicit goal in this report is to investigate to what degree the specification techniques used in the the AER/NCA protocol case study are "generally" useful and can be reused for other modelling tasks, or whether one must develop new specification techniques for each new protocol. I found that many of the techniques in the AER/NCA case study, such as e.g. the modelling of communication through link objects and the modelling of timers and timeouts, were useful also for the NORM protocol.

Chapter 2

Real-Time Maude

The *duration* of—and between—events affects not only the execution time, but also the *functionality* of many computer systems. In network communication protocols, where there may be no way of knowing whether a message will eventually arrive at the recipient, the communicating nodes use time as a means to determine their actions. A node that expects a reply to a message it sends, may set a timer upon transmitting the message. If the timer expires without the node having received a reply, it will conclude that either its own message, or the message of the other party is lost, and take the necessary action.

In general, computer systems whose functionality is dependent on time are called *real-time* systems. Other examples of such systems include control systems, monitoring systems, and multimedia systems. In particular, the behaviour of the protocol that we want to analyse—NORM—depends heavily on time. Reasoning about real-time systems poses additional requirements on the modelling language and analysis tool, requirements that extend those for systems whose functionalities are not affected by time. *Real-Time Maude* [1, 27] is a specification language and analysis tool developed by Peter C. Ölveczky and José Meseguer specifically for analysing real-time and *hybrid* systems[1]. Real-Time Maude is an extension of the specification language *Maude* [13, 14, 4]. Maude is based on *rewriting logic* [19, 9], a logic for modelling concurrent change in *distributed* systems. In the Real-Time Maude language, one can make explicit the way in which change in a system is dependent on time. A Real-Time Maude specification can be analysed using the Real-Time Maude tool by

- *simulating* one possible behaviour of the system in time,
- *searching* through all possible behaviours of the system, possibly up to a given time limit, from an initial state looking for desired or non-desired states, and
- *model checking* the system. Model checking is done by formulating properties as *linear temporal logic formulas*, and verifying that the system, w.r.t. an initial state, satisfies a property by checking that every possible behaviour of the system from the initial state

[1] A hybrid system is a real-time system where a computer program receives input from and controls physical devices.

satisfies the formula. Again, the behaviours are possibly checked up to a given time limit.

This chapter presents the Real-Time Maude language and tool. The first section describes the Maude system and its underlying logic, Section 2.2 presents the Real-Time Maude specification language, and Section 2.3 describes how Real-Time Maude specifications can be executed and formally analysed. Since the multicast protocol I will model in this thesis is a distributed system, it is convenient to use *object-oriented* specification techniques for the task, i.e., to model the protocol as a system of objects that interact by sending messages to each other. Consequently, this chapter will focus on the object-oriented features of Maude and Real-Time Maude. My goal is to make the reader able to read and understand the Real-Time Maude specification I will present in later chapters.

2.1 The Maude Language and Tool

Maude is a high-level modelling formalism for formal specification of distributed systems, based on the theory of rewriting logic. In Maude, the *static* parts of a system, i.e., the data types, are described by *equations*, and the *dynamic* parts, i.e., the system's transitions, are described by *rewrite rules*. Maude specifications are *executable*, and can be subjected to simulation and formal analysis using the Maude interpreter. Maude also has (a prototype of) syntactic support for object-oriented specification through its extension *Full Maude*. The Maude system is being developed at SRI International and the University of Illinois at Urbana-Champaign under the supervision of José Meseguer. Information about the Maude project can be found at `http://maude.cs.uiuc.edu/`.

After giving a brief introduction to rewriting logic, I will describe the basics of the Maude language, and in particular the syntax for object-oriented specification.

2.1.1 Rewriting Logic

Rewriting logic [9] is a logical framework for reasoning about a number of different computational models. In particular, rewriting logic is suitable for describing and analysing concurrent distributed systems.

Rewrite Theories

A *rewriting logic specification*, or *rewrite theory*, extends an equational specification with a set of labeled rewrite rules that model the local, atomic transitions of a system. An *equational theory* is a tuple (Ω, E), where Ω is a signature, and E a set of equations and membership axioms. The underlying equational logic which parameterises rewriting logic can be unsorted, many-sorted, order-sorted, or membership equational logic [20]. For example, an order-sorted signature is a tuple $\Omega = (S, \leq, \Sigma)$ where S is a set of sorts, $\leq$ is a partial order on the sorts, and Σ is a set of function symbols with their arity and value sorts. Given a sorted set of variables X, we denote by $\mathcal{T}_\Omega(X)_s$ the set of well-formed terms of sort s in the signature Ω.

A *rewrite theory* is a tuple $\mathcal{R} = (\Omega, E, L, R)$ where Ω and E are, as above, an equational signature and a set of equations and membership axioms, L is a set of labels, and R is a set of unconditional and conditional rewrite rules of the forms

$$l : \ \ t \rightarrow t'$$

and

$$l : \ \ t \rightarrow t' \ \ \mathbf{if} \ \ \mathit{cond}$$

where $l \in L$, t and t' are terms in $\mathcal{T}_\Omega(X)_s$ for some sort s, and *cond* is a conjunction of rewrite conditions of the form $u \rightarrow u'$, equational conditions of the form $v = v'$ and membership conditions $w : s$, for u, u', v, v', w terms in $\mathcal{T}_\Omega(X)_s$ for some sort s, and s is a sort in Ω.

Deduction Rules for Rewriting Logic

Given a rewrite theory $\mathcal{R} = (\Omega, E, L, R)$, we have that $\mathcal{R} \vdash t \rightarrow u$ if and only if $t \rightarrow u$ can be obtained in a finite number of deduction steps with the following rules of deduction:

1. **Reflexivity**:

$$\frac{}{t \rightarrow t}$$

2. **Equality**:

$$\frac{t \rightarrow t' \qquad E \vdash t = u \qquad E \vdash t' = u'}{u \rightarrow u'}$$

3. **Transitivity**:

$$\frac{t_1 \rightarrow t_2 \qquad t_2 \rightarrow t_3}{t_1 \rightarrow t_3}$$

4. **Congruence**: For each function symbol f in Ω,

$$\frac{t_1 \rightarrow u_1, \ldots, t_n \rightarrow u_n}{f(t_1, \ldots, t_n) \rightarrow f(u_1, \ldots, u_n)}$$

5. **Replacement**: For each rewrite rule $l : \ t(x_1, \ldots, x_n) \rightarrow u(x_1, \ldots, x_n)$ in $\mathcal{R}$,

$$\frac{t_1 \rightarrow u_1, \ldots, t_n \rightarrow u_n}{t(t_1/x_1, \ldots, t_n/x_n) \rightarrow u(u_1/x_1, \ldots, u_n/x_n)}$$

where $t(t_1/x_1, \ldots, t_n/x_n)$ is the term obtained by simultaneously replacing each occurence of the variables x_i in t by t_i.

Computationally, we can view the signature of a rewrite theory as describing the structure of states in a dynamic system, and rewrite rules of the form $t \rightarrow t'$ as describing atomic, local transitions in the system. *Logically*, the rules are inference steps from sentences of type t to sentences of type t'.

In addition, in *generalised rewriting logic* [9], which is supported by Maude, we can declare operators to have *frozen* argument positions, so that no rewrites can take place in these positions.

2.1.2 Maude Specifications

The Maude language supports the specification of rewrite theories. Equational theories are represented as *functional modules*, and rewrite theories as *system modules*. Maude specifications are mathematical objects that can be reasoned about formally, and at the same time they provide an executable model of a system.

Functional Modules

A functional module specifies one or more data types and operations on these, and has the following syntax:

```
fmod module name is
  module inclusions
  declarations
endfm
```

The sorts of the data types are declared with the keyword `sort`, and subsorts are specified using `subsort`. Functions are declared by `op` declarations

```
op f : s1 ... sn -> s [attributes] .
```

where f is a function symbol and $s_1, \ldots, s_n$ and s are sorts. If the number of arguments to f is zero, then f is called a *constant* of sort s. Function symbols can be declared with both prefix and "mix-fix" form. In the latter case, the positions of the arguments are given by underscores ('_') in the function declaration. A function declaration may also contain attributes that specify properties of the function, such as associativity and commutativity.

A functional module can contain equations and variable declarations. Variables are either declared separately with the keyword `var`, or within the equations. Equations may be either unconditional or conditional:

```
eq t = u .
ceq t = u if cond .
```

In addition, the module can contain unconditional or conditional membership axioms, which state that a term has a certain sort:

```
mb t : s .
cmb t : s if cond .
```

The conditions of equations and membership axioms are conjunctions of equations, boolean expressions and membership tests.

Finally, we can import other, predefined (functional) modules with the module inclusion declarations `protecting` or `including` (or the shorter versions `pr` and `inc`). Comments in a module are preceded by ***.

Equational specifications are required to be *terminating* and *confluent*. The equations of a specification are used to *reduce* a term to its *normal form*, i.e., an irreducible term of the same sort. An equational specification is terminating if its set of equations does not lead to infinite computation during reduction of terms. It is confluent if the reduction of a term always yields the same result, no matter in which order, and where in the term, the equations are applied.

System Modules

System modules are declared with the keywords `mod ... fmod`, and can, in addition to the declarations described in the previous section, contain unconditional and conditional rewrite rules of the form

```
rl [label] : t => u .
crl [label] : t => u if cond .
```

which model the transitions of a system. In addition to the conditions used in equations and membership axioms, rewrite conditions can contain rewrite tests.

Many dynamic systems are nonterminating, and/or may behave in a nondeterministic way. Unlike equational specifications, rewrite specifications can be both nonterminating and nonconfluent. However, the equational part of a rewrite specification is still required to be terminating and confluent.

2.1.3 Object-Oriented Specification in Full Maude

The current version of Maude does not provide direct support for object-oriented specification, but this is planned for future versions of the system. Meanwhile, we can use *Full Maude* [14, Part II], a prototype of support for object-oriented specification in Maude. Full Maude is a program written in Maude, which provides syntactic support for

- creating classes and objects,

- multiple inheritance,
- representing object systems as configurations of objects and messages, and
- modelling change in an object system by specifying rewrite rules on configurations.

An object-oriented Full Maude module has syntax

```
(omod module name is
  module inclusions
  declarations
endom)
```

All input to Full Maude, including Maude's functional and system modules, must be enclosed in a pair of parentheses. The Full Maude program translates an object-oriented module into an ordinary Maude module, which can then be executed by the Maude interpreter.

An object-oriented Full Maude module automatically imports the sorts `Oid` (for *object identifiers*), `Attribute` (for pairs of an *attribute name* and an *attribute value*), `AttributeSet` (for *sets* of such pairs), `Object` (for *objects*), `Msg` (for *messages*) and `Configuration` (for multisets of objects and messages), plus the associative and commutative multiset operators `_,_` for terms of sort `AttributeSet`, and `__` (with empty syntax) for terms of sort `Configuration`. In addition, the subsort declarations `Attribute < AttributeSet`, and `Object Msg < Configuration` are added to the module.

A class C is declared with syntax

```
class C | att1 : s1, ..., attn : sn .
```

where att_1 to att_n are the attributes of the objects of the class C, with values of sorts s_1 to s_n. The attribute set of a class may be empty.

An object of a class C is represented as a term

```
< O : C | att1 : val1, ..., attn : valn >
```

where O is an object identifier, att_1 to att_n are the object's attributes, and val_1 to val_n the current values of the attributes. Object identifiers, or names, are terms of the sort `Oid`. This sort has no predefined values, so we must provide values to `Oid` by a subsort declaration `subsort` *sort* `< Oid`, or by explicitly declaring such values with `op`. Since the attribute set of objects is constructed with the associative and commutative multiset operator `_,_`, the order of the attributes in an object term is not important.

Objects can communicate by sending messages to each other. Messages are terms of sort `Msg`, which are declared with syntax

```
msg m : s1 ... sn -> Msg .
```

The state of an object system is modelled as a multiset of objects and messages. We represent the system as a term of sort `Configuration` with the multiset constructor `__`. A term of sort `Msg` or `Object` is in itself a (singleton) configuration. We can also add our own elements to configurations through subsorting or by explicitly declaring a term to be of sort `Configuration`. The empty configuration is denoted by the constant `none`.

The dynamic behaviour of an object system is modelled by *rewrite rules*, which specify the possible transitions in the system. For example, in the following rule, an object `O` of class `C` receives a message `m` with address `O` and content `x`, reads the message and updates the current value of its attribute `att2` to `y + x`, stores the value `x` in `att3`, and generates a new message `m` with destination `O'` and content `y`:

```
rl [l] :
  m(O, x)   < O : C | att1 : O', att2 : y >
  =>
  < O : C | att2 : y + x, att3 : x >   m(O', y) .
```

By convention, an attribute whose value is used in the right-hand side of a rule but is not changed by the rule, such as `att1`, need only be mentioned in the left-hand side. An attribute whose value *is* changed by the rule must be mentioned in the left-hand side only if its current value is used in computing the new value, such as `att2`. Otherwise, as exemplified by `att3`, it is not necessary to mention it in the left-hand side. In short, we only have to mention those attributes of an object that affect or are changed by the application of a rule.

Objects and messages can be created or deleted by a rule. Synchronous communication can be modelled by rules where several objects interact and change state simultaneously, whereas asynchronous communication through message passing can be expressed by rules where objects send or read messages.

With subclass declarations of the form

```
subclass C < C1 ... Cn .
```

we can define a class C to be a subclass of classes C_1 to C_n. The subclass inherits the attributes and rewrite rules defined for its superclasses.

A rewrite rule in Maude roughly corresponds to a *method* in the object-oriented terminology. Due to the emphasis on generality and ease of specification, there is no support for method specialisation in the Maude language.

2.1.4 Execution and Analysis of Maude (and Full Maude) Specifications

Once we have a Maude (or Full Maude) specification of a system, we can simulate it by executing the specification with the Maude interpreter. By using the command `rew` we can explore *one* of the possibly many different rewrite sequences from an initial state. The Maude rewrite engine uses equational simplification to reduce the terms of the specification to their normal forms, and performs rewrites on these normal forms (i.e., the resulting term is normalised after each application of a rewrite rule) as long as any rewrite rule applies. If a specification is nonterminating, it can be executed with a bound on the number of rewrites to be performed. The Maude interpreter will then apply the rules until it reaches the user-specified bound, and return the resulting state of the system.

A specification can be further analysed with the Maude tools for search and model checking. Using the command `search`, *all* the possible behaviours of a system from an initial state can be explored in search for states matching a given search pattern. The Maude system is also equipped with a linear temporal logic model checker which can be used for analysing properties of finite-state systems.

In addition to the built-in tools for rewriting, search and model checking, a user can take advantage of Maude's *reflective* capabilities [14], and specify her own analysis strategies directly in the Maude language.

2.2 Object-Oriented Specification in Real-Time Maude

In [25], Ölveczky and Meseguer showed that real-time and hybrid systems can be modelled using rewrite theories by including a data type and appropriate equations which model a time domain, and by making explicit the duration information for rewrite rules that model the elapse of time in a system. Such rewrite theories are called *real-time rewrite theories*. In the same way that ordinary rewrite theories can be specified in the Maude language, real-time rewrite theories can be specified in Real-Time Maude. In addition to providing syntactic support for specifying real-time systems, Real-Time Maude is equipped with a set of commands for "real-time specific" execution and analysis of Real-Time Maude specifications. Real-Time Maude is built on top of the Full Maude extension of Maude, and lets the user add time to both ordinary modules and object-oriented modules.

This section presents the Real-Time Maude specification language, and will focus on its object-oriented features. Section 2.3 presents the Real-Time Maude tool. The presentation is based on the Real-Time Maude 2.0 Manual [22], which can be downloaded from the Maude web page.

2.2.1 Real-Time Rewrite Theories

This section describes real-time rewrite theories without going into details on the theoretical aspects, which can be found in [25]. A real-time rewrite theory is a rewrite theory which contains:

- A specification of a sort `Time` for the time domain, which may be discrete or dense, and which satisfies the axioms of the theory *TIME* described in [25].
- The data sorts `GlobalSystem`, which has no subsorts or supersorts, and `System`. The sort `System` denotes the state of a system. A free constructor

  ```
  {_} : System -> GlobalSystem .
  ```

  takes as argument the state t of a system and returns a term $\{t\}$ of sort `GlobalSystem`, which denotes the *whole* system in state t.
- Ordinary rewrite rules that model *instantaneous* change in a system, i.e., the local concurrent transitions of the system that are assumed to take zero time.
- *Tick rules*, i.e., rewrite rules that model the elapse of time in a system:

  $$l : \ \{t\} \xrightarrow{\tau} \{u\} \ \ \textbf{if} \ \ cond$$

  A tick rule rewrites a system in state t to the state u given that the conditions in *cond* hold. The term τ of sort `Time` denotes the *duration* of the rewrite. Tick rules must only be applied on the system as a whole, i.e., on terms of sort `GlobalSystem`, to ensure that time advances uniformly in all parts of the system.

2.2.2 Timed Modules and Object-Oriented Timed Modules

Real-time rewrite theories are specified in Real-Time Maude as *timed modules* and *object-oriented timed modules*. Since Real-Time Maude is built on top of Full Maude, all input to Real-Time Maude must be enclosed in a pair of parentheses. Timed modules and object-oriented timed modules are enclosed by the keywords `tmod ... endtm` and `tomod ... endtom`, respectively. Timed and object-oriented timed modules automatically import declarations of the sorts `System` and `GlobalSystem`, and the operator `{_}`.

A subsort declaration `Configuration < System` is automatically added to object-oriented timed modules, thus specifying the state of a system to be a configuration of objects and messages. Real-Time Maude extends the sort `Configuration` to provide a richer sort structure by including the following sorts in every object-oriented timed module,

```
sorts EmptyConfiguration NEConfiguration MsgConfiguration
      NEMsgConfiguration ObjectConfiguration NEObjectConfiguration .
```

for empty configurations, nonempty configurations, configurations consisting only of messages, nonempty message configurations, configurations consisting only of objects, and nonempty object configurations, respectively.

2.2.3 Time Domains

Different modelling tasks may require using different time domains, and from a user perspective it is a great advantage to be able to freely choose an appropriate time domain for a specification. Instead of restricting the user to choose from a set of predefined time domains, Real-Time Maude provides a framework for specifying time domains, which the user can complete according to her needs. Every timed module and object-oriented timed module automatically imports the module `TIME`, which declares a sort `Time` of time values and defines a constant `zero` of sort `Time`. It also declares a set of time operations: `_plus_` and `_monus_` for addition and cut-off subtraction, and `_le_` (less or equal), `_lt_` (less than) `_ge_` (greater or equal) and `_gt_` (greater than) for comparing time values. The user can define a time domain by

- specifying values for the sort `Time` by importing an appropriate data type and adding a subsort declaration *data type* `< Time`,
- and by giving appropriate equations interpreting the constant `zero` and the operators `_plus_`, `_monus_` and `_lt_` relative to the chosen data type (equations for the other operators are already given in the module `TIME`).

In addition to allowing the user to define her own time domain, Real-Time Maude provides some built-in time domains that can be imported into a specification. The module `NAT-TIME-DOMAIN` gives a discrete linear time domain of the natural numbers, and `POSRAT-TIME-DOMAIN` defines a dense linear time domain of the positive rational numbers. It is often useful for specification purposes to have an "infinite" time value. In Real-Time Maude, this is provided by an extention of the time domain framework with an infinity value `INF`:

```
sort TimeInf .    subsort Time < TimeInf .

op INF : -> TimeInf .
```

The built-in modules `NAT-TIME-DOMAIN-WITH-INF` and `POSRAT-TIME-DOMAIN-WITH-INF` extend the predefined time domains with this infinity value, and may be imported into a specification.

2.2.4 Time Advance in a Configuration

In Section 2.1.3, we saw how Maude's rewrite rules can be used to specify the local transitions in an object-oriented system. In Real-Time Maude, such rewrite rules are called *instantaneous rewrite rules*, and are used to model instantaneous actions that are assumed to take zero time. In addition to rules specifying local change in a system, a Real-Time Maude specification will also contain rules that specify how the *whole* system changes as time advances. Such rules, called tick rules, have syntax

```
rl [l] : {t} => {u} in time τ .
```

for unconditional rules, or

```
crl [l] : {t} => {u} in time τ if cond .
```

for conditional ones. The rules say that the global system in state t is rewritten to state u in time τ. The kind of changes modelled by such rules might for example be the updating of local clocks and timers in the objects in a configuration. Since time is something that affects *all* components of a system simultaneously, the tick rules always rewrite terms of sort `GlobalSystem`. If we were to allow timed rewrites on terms of sort `Configuration` (for object-oriented specifications), we would run the risk of time advancing on only a *part* of the system, since the tick rule might be applied on a subterm of the term representing the configuration.

How much time should pass in a system when a tick rule is applied? For some modelling tasks, it is appropriate to model a system with *deterministic* time increase. The digital clock on my computer screen will for example change state once every minute, and it would be natural to model time increase in such a clock by a tick rule where τ is given a fixed value of 1 time unit, representing one minute. In other systems, we may not know how much time will pass before a state change takes place. In such cases, it is more appropriate to have a *nondeterministic* tick rule, where τ is a variable which is instantiated with a time value when the rule is applied during execution of the specification. Unlike deterministic tick rules, nondeterministic tick rules are not directly executable in Maude, because it is not clear how to instantiate the time variable occuring in the right hand side of the rule. However, Real-Time Maude has some built-in strategies which make it possible to execute nondeterministic rules.

2.3 Execution and Analysis of Real-Time Maude Specifications

Real-Time Maude extends Maude's rewriting, search and model checking capabilities to provide a set of tools specifically for executing and analysing timed specifications. This section gives an overview of the syntax and use of the various Real-Time Maude commands.

Real-Time Maude takes as much advantage as possible of the powerful Maude rewrite engine by translating Real-Time Maude modules into Maude modules and executing them using the Maude interpreter. The majority of the Real-Time Maude commands execute such translated modules with the Maude interpreter, and are therefore very efficient.

Section 2.3.1 describes the Real-Time Maude rewrite command; Section 2.3.2 describes the various search commands; and Section 2.3.3 explains how Real-Time Maude specifications, both finite-state and infinite-state, can be model checked. In the previous section, I mentioned that it is possible to have both deterministic and nondeterministic tick rules in timed specifications, and that the latter type is not directly executable because the duration of the rewrite is denoted by a by a *fresh* variable which does not occur in the left-hand side of the rule. However, nondeterministic tick rules can be made executable by choosing one of several built-in strategies for instantiating the time variable at execution time. I use a deterministic tick rule in my model of the NORM protocol, because, in NORM, every instantaneous action is triggered by either the expiration of a timer, or by the arrival of a message. Due to this choice, and because this is just an overview of the Real-Time Maude tool, I will not describe the strategies for executing nondeterministic tick rules, but rather refer the reader to the Real-Time Maude manual [22].

2.3.1 Timed Rewriting

As I explained in Section 2.1.4, the Maude `rew` command simulates *one* possible behaviour of a system by applying the rewrite rules from an initial state. In Real-Time Maude, the command for simulating one behaviour is called `trew`. Like Maude's `rew` command, it can be used with or without a bound on the number of rewrite steps to be performed. In addition, we can choose whether we would like to have a *time bound* on the rewrite such that the command stops before or at a given time limit and returns the resulting state of the system. If we do not specify a time bound, the rewrite continues either until it reaches a state which cannot be further rewritten, or indefinitely, if the specification is nonterminating. The state returned by the command is equipped with a time stamp which shows the total duration of the rewrite sequence.

If we wish to use `trew` without specifying a time bound on the rewrite, the command has the following syntax:

```
(trew [[n]] initState with no time limit .)
```

The number n denotes the upper bound on the number of rewrite steps that can be performed and is optional. *initState* is the initial state of the system, and is a term of sort `GlobalSystem`.

The syntax for the `trew` command *with* a time bound on the rewrite is as follows,

```
(trew [[n]] initState in time op timeLimit .)
```

where *op* is either of the operators `<=` and `<`, and *timeLimit* is a time value from the specified time domain.

2.3.2 Search

Although it is valuable to be able to simulate *one* behaviour of a system for initial prototyping purposes, one may analyse *all* possible behaviours in order to gain further insight into the system. Real-Time Maude's search commands make it possible to analyse all behaviours of a system w.r.t. an initial state by searching for states that match a given search pattern. The search pattern may characterise both desired and non-desired states that might occur in the system.

The Real-Time Maude search commands tsearch and utsearch take advantage of Maude's poweful search tool to provide *timed* and *untimed* search in timed specifications. The *timed* search command tsearch lets the user search for states that can be reached from an initial state within a given time interval. With a specified time *limit*, the search is guaranteed to terminate (unless the specification exhibits "Zeno behaviour"). If the search is successful, a set of solutions matching the search pattern is returned, along with time stamps showing the time it takes to reach the states. Otherwise, Real-Time Maude just returns "no solution". There are three versions of the tsearch command

```
(tsearch [[n]] initState =>* searchPattern with no time limit .)
(tsearch [[n]] initState =>* searchPattern in time op timeLimit .)
(tsearch [[n]] initState =>* searchPattern
         in time-interval between op timeLimit and op′ timeLimit′ .)
```

The number n denotes an upper bound on the number of solutions the command is to search for, and is optional. The initial state *initState* and the search pattern *searchPattern* are terms of the sort GlobalSystem. The arrow =>* means that the tool searches for states that are reachable in *zero or more* rewrite steps from the initial state. The arrow can be replaced by =>+ or =>!, for search for states reachable in *one or more* steps, or *deadlocked* states (states that cannot be further rewritten), respectively. The last two commands can be used to search for states that can be found within a time interval, where *op* and *op′* are either <, <=, >, or >=, and *timeLimit* and *timeLimit′* are time values.

The tsearch command can be very useful for analysing infinite-state systems, because the state space can be restricted with a time bound so that the reachable state space becomes finite. For systems with a finite state space, Real-Time Maude provides the search command utsearch for *untimed* search. Whereas tsearch considers two states that are otherwise equal to be different because they have different time stamps (i.e., occur at different points in time), the utsearch command does not take time into consideration when looking for states that match the search pattern. The syntax of the untimed search command is

```
(utsearch [[n]] initState =>* searchPattern .)
```

where n, *initState*, =>* and *searchPattern* have the same meaning as for the timed search command, and =>* can be replaced by =>+ or =>!.

In addtition to the tsearch and utsearch commands, Real-Time Maude has two search commands for finding the shortest and longest time it takes to reach a state matching a search pattern:

```
(find earliest initState =>* searchPattern .)
(find latest initState =>* searchPattern with no time limit .)
(find latest initState =>* searchPattern in time op timeLimit .)
```

The `find earliest` command returns the first occurence of a state which matches the search pattern, i.e., the state which took the *shortest* time to find. The command `find latest` looks at all behaviours from the initial state in search of the one where it takes the *longest* time for a match with the search pattern to occur. It then returns the matching state from this behaviour. If there is a behaviour where there is no match with the search pattern, the command returns a negative answer.

2.3.3 Temporal Logic Model Checking

The Maude tool comes with a high-performance temporal logic model checker for finite-state systems [15]. In the Real-Time Maude tool, this model checker is extended to provide *timed* and *untimed* model checking of Real-Time Maude specifications. The idea is the same as for the timed and untimed search commands: In timed model checking one can model check infinite-state systems by limiting the state space with a time bound on the behaviours. If the system has a finite state space, one can use untimed model checking, which does not associate the states with a time stamp.

The user defines temporal properties as formulas in propositional linear temporal logic. The temporal language is made up of

- atomic propositions,
- the constants `True` and `False`,
- the logical operators `~` (negation), `/\` (conjunction), `\/` (disjunction), `->` (implication) and `<->` (equivalence), and
- a set of temporal operators, such as `U` (“until”), `<>` (“eventually”), `[]` (“always”), etc.

The constants `True` and `False`, the logical operators and the temporal operators are predefined. The atomic propositions must be defined by the user as terms of sort `Prop`. The temporal formulas constructed from these components are of sort `Formula`.

Given a specification of a system in a module *SYSTEM*, the user defines a module which imports *SYSTEM* and the module `TIMED-MODEL-CHECKER`, which contains the Real-Time Maude extension of the Maude model checker. The user also defines one or more atomic propositions and gives appropriate equations that define the states for which a proposition holds, with the following syntax:

```
(tomod MODEL-CHECK-SYSTEM
  including TIMED-MODEL-CHECKER .
  protecting SYSTEM .

  op atomicProp :  -> Prop [ctor] .

  eq statePattern |= atomicProp = booleanExpression .
endtom)
```

The atomic propositions may be parameterised, i.e., may have a list of arguments, and *booleanExpression* is a term of sort `Bool`.

The specified system can then be model checked using either the *timed* model checking command

```
(mc initState |=t formula in time op timeLimit .)
```

or the *untimed* model checking command

```
(mc initState |=u formula .)
```

where *formula* is a term of sort `Formula`. The model checking commands return `true` if the temporal property expressed by *formula* holds in the system, and return a counterexample otherwise.

Chapter 3

NORM - A Multicast Transport Protocol

This chapter presents the multicast transport protocol NORM [7]. After a short introduction to multicast transport protocols in Section 3.1, the rest of the chapter explains the purpose of the NORM protocol, and how it is supposed to be achieved.

3.1 Multicast Transport Protocols

In computer networks, we distinguish between three main forms of communication:

- *unicast*, where a sender wants to transmit messages to *one* receiver,
- *broadcast*, where a sender wants to transmit to *all* the other nodes, and
- *multicast*, where a sender wants to transmit to a *subgroup* of the receivers connected to the network.

In multicast communication, a sender send its messages to a specific *multicast address*, and it is up to the *network* to make sure that every member of the group defined by the address gets a copy of the messages.

The Internet does not yet provide multicast communication to all its users. However, a growing number of networks connected to the Internet offer multicast. This opens the way for developing more applications that presuppose Internet multicast communication. Multicast-based applications cover a wide range of categories, from chat groups and multi-player games, to multimedia conferencing tools and stock price subscriber services [29].

Different types of multicast applications have different kind of requirements to the communication services provided by the network. For example, a multimedia conferencing application that makes it possible for users to see and talk to each other in real time is sensitive to *delay*, i.e., the time it takes to send data packets to the receivers. The application can tolerate some delay of video data, but the voices of the participants become unintelligible if the audio data suffers large variances in delay. If a data packet gets lost once in a while, the tool still functions quite well. However, the data that does arrive must arrive on time. On the other hand, an application that allows several users to edit shared documents requires *reliable* data delivery, i.e., that every data packet arrives, that they arrive in the same order they were sent, and that only one copy of each packet is delivered.

The problem is that the Internet cannot provide neither timely nor reliable delivery of data between applications. The Internet Protocol (IP), which ties together networks with different technologies, provides a *best effort* service: It will do its best to deliver data packets to the recipients, but it makes no guarantees about the delivery. Data packets may be reordered or duplicated, and they may arrive late, or not arrive at all.

The multicast protocol that this chapter describes—NORM—is an example of how this problem is handled, both in the Internet and in networks in general: A set of services that are useful to a group of applications are identified and implemented in a *transport protocol*, a protocol which uses the underlying network's packet delivery services to provide communication between application programs on computers attached to the network [28]. The protocol then tries to provide the services despite the limitations of the network, by using various techniques for *preventing*, *detecting*, and *correcting* errors. Below are some examples of techniques that a transport protocol can use:

- **Preventing errors:**
  - If the network is congested, the sender can lower its transmission rate to try to prevent further packet loss.
  - The receivers suppress superfluous feedback messages to keep the amount of feedback to the sender at a minimum. This helps to reduce network congestion, which often leads to increased packet loss.
- **Detecting errors:**
  - The data packets are numbered, so that receivers can detect gaps in the transmission.
  - The receivers send acknowledgment messages to the sender for the data they have received, or, alternatively, non-acknowledgments for data which has not arrived or which is damaged.
- **Correcting errors:**
  - The sender retransmits data packets that have been lost or damaged.

The design of a multicast transport protocol often has to balance the use of feedback and retransmission techniques with a requirement for *scalability*, i.e., that the protocol is able to scale to a large number of users. In addition, while trying to fulfil the requirements from applications, a protocol has to make sure that it does not use more than its fair share of the network resources. This is often referred to as *TCP-friendliness*, because protocols will have to share resources with TCP, the major transport protocol in the Internet.

3.2 An Overview of the NORM Protocol

The specification of the *negative-acknowledgement oriented reliable multicast* (NORM) protocol is part of the Internet Engineering Task Force's (IETF) ongoing work of standardisation of Internet protocols. NORM is defined in an Internet-Draft by B. Adamson, C. Bormann, M. Handley and J. Macker [7]. An IETF Internet-Draft is a working document valid for a period of six months at a time, and which is replaced by a request for comments (RFC) when the standardisation work is completed. The NORM draft that I describe in this chapter was issued in March 2003 and expired in September 2003. Some of the NORM algorithms are also described in another document by the authors, the NORM building blocks document [6], which specifies a set of general building blocks that can be used in constructing a working protocol . The building block document also expired in September 2003. Both documents have since been replaced by updated versions, which can be found at the IETF web site at `http://www.ietf.org/html.charters/rmt-charter.html`. I wish to emphasise that the NORM draft and the building block specification represent work in progress.

The NORM protocol is designed to provide *reliable*, *efficient*, *scalable*, and *robust* transport of large amounts of data over an IP multicast network. The protocol uses

- a *NACK-based repair strategy*, where the receivers request repair of missing data by sending non-acknowledgments (NACKs) when they discover packet loss,
- *receiver feedback suppression mechanisms*, that allow the receivers to cancel superfluous feedback messages to the sender, and
- *congestion control*, so that the receiver can adjust its transmission rate according to the network conditions.

The motivation for using negative acknowledgments is to keep the amount of feedback from receivers as low as possible. The less feedback a protocol requires, the more scalable it can be. To further reduce feedback from receivers, NORM lets every receiver multicast its feedback messages to the whole group. A receiver who needs repair (or is about to send some other feedback message to the sender) can listen to the messages multicast by the receivers, and cancel its message if some other receiver has already asked for repair of the same packets.

An important part of the NORM algorithms are the various *timeouts* used by senders and receivers. The timeouts are all based on the group's *greatest round-trip time* (GRTT). The round-trip time (RTT) of a protocol participant is the time it takes for a message to travel from that node to the other end of the network and back. The GRTT value is an approximation of the largest receiver RTT value in the group.

In addition to a NACK-based repair strategy and feedback suppression, *forward error correction* (FEC) techniques can be used with NORM to further reduce the amount of repair requests and repair transmissions. FEC is a general strategy for recovering from errors by including redundant information in each data packet. The additional information makes it possible for the receiver to discover bit errors in the data content and reconstruct the correct data [28]. Since this is optional in NORM, and since the use or omission of FEC does not influence the logic of the protocol, this chapter (and the model I will develop in subsequent chapters) will not describe how FEC is used in NORM.

After a brief look at the multicast service offered by IP, and a few words on how data content is identified in NORM, the remainder of this section describes the protocol's messages and algorithms. The model I present in subsequent chapters focuses on data and repair transmission, and GRTT measurement. The following overview of NORM concentrates on these components, and congestion control will only be mentioned briefly. The details can be found in Appendix A, which contains the entire NORM protocol specification.

3.2.1 IP's Multicast Service

The NORM protocol uses the delivery services of IP, a protocol that makes it possible for application processes to communicate across networks of different technologies. Because IP connects very different kinds of networks, its delivery service is kept as simple as possible: IP

- assigns a unique *address* to every host (machine) in the Internet, and
- delivers a data packet encapsulated in a *datagram*.

IP does not try to handle the complexity in transmitting a series of data packets that form some sort of unit. Instead, it treats every packet as a separate message, a datagram, which is sent independently of other packets. Each datagram contains the full address of the destination network and the receiver on that network. It is up to the nodes that tie two or more networks together, the *routers*, to decide how to forward each packet to the next hop along the path. As a consequence, two packets that form a unit may be sent to their destination along different paths (as network conditions change).

IP makes multicast in the Internet possible by providing special addresses that denote multicast groups. A node that wishes to join a multicast group, notifies the closest router of this. The router will then forward every datagram addressed to that particular group to the node. The node can also send its own messages to the group's address.

3.2.2 Identification of Data Content

Many applications require that the transport service allows the application to send arbitrarily large messages [28]. Since the payload of the data packets that a protocol sends out over the network has a limited size, the protocol solves this problem by *fragmentation* and *reassembly*: Messages that are too large to fit in the payload of a data packet are fragmented into smaller units, *segments*, and sent across the network. At the receiver, the segments are reassembled into the original message. To make it possible for the receiver to reassemble a message, there must be some kind of identification scheme, which numbers the segments and identifies the message to which they belong.

In NORM, such messages are called *objects*. NORM objects can be finite size messages, or they can be "infinite" streams of data. In either case, an object in transit is identified with an object number[1]. Each segment of an object is identified with the number of the object to which it belongs, and a sequence number for the segment. These identifiers are included in every packet that carries a data segment. To make things simple, I call such segment identifiers "sequence numbers" throughout the rest of this chapter.

3.2.3 Protocol Messages

This section lists the main message types of the NORM protocol and describes their function.

Sender Messages

- *NORM_DATA*: This message type is used both for sending new data segments, and for retransmitting segments requested by receivers. When the message is used for repair transmission, a flag is set to inform the receivers that this is a retransmission.

- *NORM_CMD*: The sender uses messages of this type to perform a number of protocol operations in order to control the transmission.

  - *NORM_CMD(CC)*: This message is used to collect round-trip time values, and to provide congestion control information to the receivers.
  - *NORM_CMD(FLUSH)*: The sender uses this message to inform the receivers that it has finished transmitting its enqueued data and repair segments, and that these will be flushed (deleted). The receivers are expected to check their repair needs up to and including the sequence number given in the message, and ask for repair if data content is missing.
  - *NORM_CMD(SQUELCH)*: If a sender receives a repair request for data content it is no longer supporting, it informs the receivers of which data objects are valid for repair by multicasting a NORM_CMD(SQUELCH) message.

Receiver Messages

- *NORM_NACK*: This message is used to request retransmission of lost or damaged segments. A NORM_NACK, i.e. a non-acknowledgement message, identifies the segments that have *not* been received, and thus implicitly acknowledges receipt of the rest of the transmitted data up to a given sequence number.

- *NORM_ACK*: Messages of this type are generated in response to various commands from the sender.

[1] The object identifier is actually a pair of identifiers: An object number, and a temporary name for the sender, to avoid confusion in cases where there are several NORM senders transmitting simultaneously.

3.2.4 Data Transmission and NACK-Based Repair Strategy

Data and repair transmission in NORM can be divided into three components [6]:

1. Sender transmission,
2. Receiver repair request process, and
3. Sender NACK processing and repair response.

Sender Transmission

The goal for the NORM sender is to transmit one or more data objects to a group of receivers on behalf of its application. During ordinary transmission of new data content, the sender performs the following actions:

- Upon startup, the sender begins sending NORM_CMD(CC) messages to collect round-trip time values and congestion control information from the receiver group.
- Then the sender either
  - proceeds with data transmission immediately, or
  - waits for feedback from the group to make sure that there are receivers present.
- During data transmission the sender segments data from the application and transmits the segments in NORM_DATA messages at a rate controlled by the congestion control mechanism.
- When the sender has finished transmitting its enqueued data content and any pending repairs, it sends a series of NORM_CMD(FLUSH) messages to inform the receivers that the data will be flushed. Receivers who need repair, respond with a NORM_NACK message. The sender transmits the requested repair segments and repeats the flush process. If the sender does not have any more new data content to send, it sets an end-of-transmission flag in the NORM_CMD(FLUSH) message to inform the receivers that it is preparing to terminate transmission.

Receiver Repair Request Process

The job of the NORM receiver is to receive data segments and pass these on to its application. If the receiver discovers a gap in the sequence numbers of the segments, it initiates its repair request procedure:

- The receiver can only initiate a NACK cycle
  - at an object boundary, or

  - when it receives a NORM_CMD(FLUSH) message

- The NACK cycle begins with a random backoff timeout, during which the receiver accumulates NACK messages from the other receivers. When the timeout expires, the receiver generates a NACK message to request repair for the missing data content only if

  1. the earliest sequence number for which the receiver needs repair is smaller than the sequence number of the last segment it has received from the sender (if not, the sender has started retransmitting segments in response to repair requests from other receivers, and the segments the receiver needs may be on their way), and
  2. the receiver's repair needs are not covered by the accumulated repair requests from other receivers.

- After sending a NACK message, the receiver must wait some time before it can initiate a new NACK cycle, to make sure the sender has time to receive, process and respond to the repair request.

A NACK message identifies missing segments, starting with the lowest sequence number up to the sequence number of the most recently received segment at the time the NACK cycle was initiated. Only one NACK message is generated per cycle, and the message content is truncated if it exceeds the message payload size.

Sender NACK Processing and Repair Response

When it receives a NORM_NACK message, the sender initiates its repair procedure. Instead of treating each repair request separately, the sender accumulates repair requests in order to make the repair transmission more effective. The sender performs the following actions:

- When it gets a repair request, the sender sets a timer. During the timeout, it aggregates NACK messages from the receivers, while continuing the transmission of new data content.

- When the timer expires, the sender stops transmission of new data content and starts to retransmit the requested segments. When it has completed repair transmission, it continues transmitting new segments.

- After the NACK aggregation timer has expired, the sender must wait some time before it can initiate a new repair cycle. If a NACK message arrives during this "hold-off" timeout, the sender includes the requested segments in its repair transmission if their sequence numbers are greater than the sender's current transmission position. Otherwise, the NACK message is ignored. Once the "hold-off" timeout is over, the sender is free to start a new repair cycle.

- If the sender receives a repair request for segments it no longer buffers, it generates a NORM_CMD(SQUELCH) message to inform the receiver set of which data objects are valid for repair.

3.2.5 Round-Trip Time Collection and GRTT Measurement

As mentioned previously, the protocol participants need a common time value to base their timeout on. The sender is responsible for calculating this value—the group's greatest round-trip time—and for advertising it to the receivers.

- The sender periodically multicasts NORM_CMD(CC) messages carrying a timestamp recording the time the message was sent relative to the sender's internal clock.
- When a receiver gets a NORM_CMD(CC) message, it stores the sender's timestamp and records the time it received the message. The next time the receiver generates a feedback message, it includes an adjusted version of the timestamp, increased with the amount of time the receiver held the timestamp. For example, we have a sender and a receiver whose clocks are synchronised: The receiver receives a timestamp from the sender—5— at time 18, and generates a feedback message at time 23. The receiver adjusts the timestamp to $5 + 5 = 10$.
- The sender collects the adjusted timestamps and calculates each receiver's RTT by subtracting the adjusted timestamp from the current time. If it receives the adjusted timestamp 10 from the example above at time 36, then the RTT for that receiver is $36 - 10 = 26$. The GRTT estimate is updated using the greatest RTT value measured during each probing interval.

The current GRTT estimate is advertised in all the sender messages. However, if the time interval between the NORM_DATA messages is larger than the current GRTT estimate, this value is announced instead.

3.2.6 Congestion Control

Packet loss in a network is often the result of *congestion*: The routers, who are responsible for forwarding packets from one network to another, receive more packets than they can handle. If the network is congested, a sender can help to reduce the damage by lowering its transmission rate. The mechanism it uses to adjust its rate is called *congestion control.* By monitoring the round-trip time values of the receivers and how much packet loss they experience, a sender can calculate an running estimate of the "ideal" transmission rate. Internet protocols are required to have congestion control.

The RTT collection algorithm that I described in the previous section is part of NORM's congestion control procedure. To reduce congestion, the NORM sender

- periodically sends out NORM_CMD(CC) messages to get RTT and packet loss feedback from the receivers,
- uses this information to identify the *current limiting receiver* (CLR), i.e, the receiver which at the moment has the largest RTT and experiences the most severe packet loss, and

- adjusts its transmission rate according to the situation of the CLR.

The loss experienced by the CLR reflects the situation in the most congested path in the multicast topology.

Chapter 4

Modelling and Analysing NORM

The previous chapters introduced the language and tool I will apply, and the multicast protocol specification I will formalise. The remaining chapters of this thesis will present my formal specification and analysis of the NORM protocol.

I start this chapter by discussing some of the preliminaries on modelling the NORM protocol. I will explain the selection I have made of protocol components that will be modelled, the structure of the model, and other general modelling choices. Finally, I will present the part of the model which describes the basics of a network and how it is affected by time.

Chapters 5 through 8 present the Real-Time Maude specifications of the NORM components and an analysis of the specifications. The modelling process lets us identify any ambiguities or inconsistencies in the informal NORM specification. If the formal analysis of the Real-Time Maude specifications reveals any errors in the protocol, we should be able to trace their origin back to the informal specification. Chapter 5 presents a formal specification of how the sender calculates and advertises the GRTT estimate, and in Chapter 6 I execute and analyse the specification. The presentation of the data content and repair transmission component follows the same pattern: in Chapter 7, I formally specify the transmission of data content and the NACK procedure, and in Chapter 8, I execute and analyse the specification.

In Chapter 9, I briefly outline how to combine the Real-Time Maude specifications so that the NORM protocol can be analysed as a whole. In the final chapter, I discuss and summarise the results from the modelling and analysis processes.

4.1 From Informal Specification to Executable Model

Before attempting to specify NORM in Real-Time Maude, we should make some decisions about what it is our model should include and what we can leave out. When making such decisions, we must ask ourselves what the model should be about: What are we interested in capturing in the model and examining with the analysis tools?

In our case we wish to capture the *logic* of the communication protocol that we model. I therefore abstract away from the implementation details of the informal specification and technicalities of network communication such as routing etc., and keep the essence of the ideas on communication that form the foundation of the technical specification.

In this section, I explain some of the choices that I made at the start of the modelling process. The choices concerning details in modelling the components of NORM will be dealt with in the corresponding chapters.

4.1.1 Selection of Procedures From the NORM Specification

In this section, I explain which functionalities of the NORM protocol I have chosen to model. Furthermore, I explain some choices I made concerning the multicast communication, and the decision to leave out forward error correction in the model.

RTT Collection and Data and Repair Transmission

The NORM draft specification describes a collection of messages and procedures whose purpose is to provide scalable end-to-end reliable transport of bulk data objects or streams over generic IP multicast services. To formally specify the whole NORM protocol is a large and complex task. Therefore I have chosen to model two components of the NORM protocol:

1. *The round-trip time component*, which produces the RTT value required to support NORM's receiver feedback suppression algorithms, and
2. *the data and repair transmission component*, which implements the service that the NORM protocol offers.

Including the congestion control component would have made the model more satisfying, however, it is beyond the scope of this thesis to model the whole NORM protocol. Leaving it out will not affect our understanding of the protocol, since the purpose of congestion control is not to ensure reliability, but to enable the sender to avoid contributing to congestion of the network.

Further Modelling Choices

While the design of the NORM protocol is principally based on the assumption of one sender transmitting to a group of receivers during a NORM session, it is also possible to have several senders transmitting simultaneously to the receivers within the same session. My model will concentrate on the *one-to-many* scenario. The NORM draft specifies reliable transmission procedures both for topologies where multicast services are available to all the members of a multicast group, and topologies where only the sender can multicast messages and the receivers provide unicast feedback. The network in my model provides *reciprocal multicast*.

The motivation for modelling only one-to-many transmission and reciprocal multicast is that the alternatives seem to be viewed as exeptions to the expected use of the NORM protocol and that the procedure descriptions do not deal with them in detail.

A consequence of these two choices is that a message does not have to carry information about who sent it. Since there is only one sender, it is not necessary for the receivers to know the identity of that node—there will be no confusion because there is only one "stream" of messages carrying data content. To the sender, it is not important *which* one of the receivers that lost a packet—all its repair messages will be multicast anyway.

I mentioned in Chapter 3 that the data content objects that can be transmitted by a NORM sender can be both finite size messages, and "infinite" streams of data. Since finite size messages is the default, this is what I use in the model. I also mentioned that the NORM protocol can use forward error correction (FEC) techniques as a means to detect and correct errors. For our purpose of understanding the protocol's communication strategies, nothing is gained by including the FEC aspect in the model, which would become unnecessarily complex and difficult to read.

4.1.2 The Structure of the Model

Instead of having one monolithic executable specification which captures every aspect of the NORM protocol, it would be a good idea to have a separate specification for each component—we could then analyse the functionality of a single component without having to look at the rest of the system. In addition, it should be possible to combine the specifications of the components so that the *whole* NORM protocol can be analysed.

The *structure* of the model I present is borrowed from Ölveczky's case study of the AER/NCA protocol suite for reliable multicast in active networks in [21]. Each component of the protocol is specified separately, and by using object-oriented inheritance techniques, the specifications can be combined into an overall model of the system. For each component, I specify the rules necessary for a stand-alone execution. Some rules, which correspond to actions in NORM which concern more than one of its components, must be redefined for the specification of the combined protocol. For instance, a message may be used to perform several functions in the system. In the specification of a single component, a rule will only reflect the functions the message has in *that* part of the protocol. For the combined protocol I will have to define new rules which model *all* these different functions. However, most of the rules specified for the subprotocols can be reused in the combined protocol.

The difficult task in the modelling process consists of turning the NORM specification's informal description of procedures into sets of Real-Time Maude rewrite rules. Unlike the AER/NCA specification [21, Appendix B], which is written in a use case-like style and divides the protocol into four subprotocols, there is no clear subdivision of the NORM protocol that immediately can serve as a template for a model. In Ölveczky's model, the rewrite rules are modelled on the use cases of the specification. One of the challenges in modelling NORM is to find sensible ways of capturing a part of the functionality of a procedure in a rewrite rule.

Figure 4.1 shows the general structure of my NORM model. A set of classes and rules used by both the separate components and the combined protocol is defined in the top layer. The

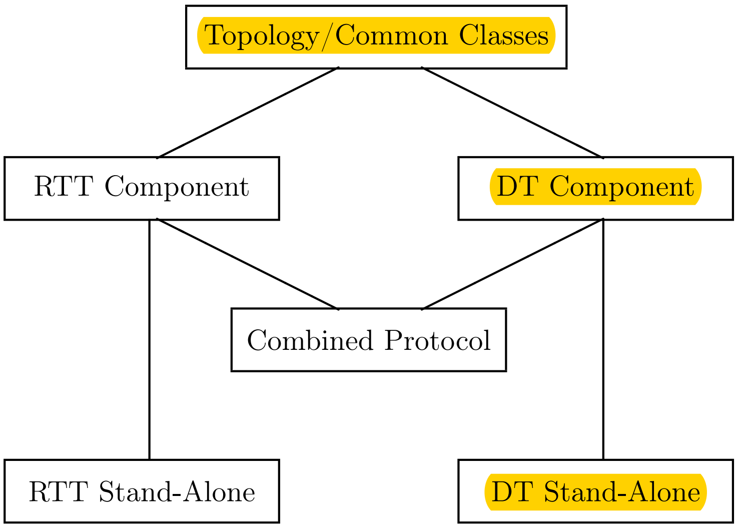

Figure 4.1: The structure of the NORM model

next layer defines classes and rules for the RTT and data transmission components that can be reused for the combined protocol. The rules in the bottom layer are specific to the stand-alone subprotocols and must be redefined for the combined specification. The class hierarchy for sender and receiver objects will be described in Section 4.2.5.

Comments on the Rewrite Rules

One of the characteristic features of network systems is that the communication is *asynchronous*. When a message is sent from one node to another, it takes a certain time before it arrives and the corresponding action is performed at the receiver. Real-Time Maude lets us model this in a natural way. Most of the rules in our model will be asynchronous.

A formal specification of the NORM protocol will make explicit hidden assumptions and knowledge in the informal specification. As a consequence, some of the rewrite rules will model aspects of NORM that are explicitly stated, and some will reflect the unstated parts of the protocol draft.

Comments on the Topology

The NORM building blocks document [6] discusses strategies for dealing with dynamic multicast groups, i.e., how to handle receivers joining and leaving the group during a session. In the present work, I will make the assumption that the network topology is *static*—no objects are created or destroyed during an execution of a specification.

Because I want to abstract away as much as possible from details that do not affect the logic of the communication, the distinction between different *network layers* in the model is intentionally blurred. For example, a NORM message travelling through a link is given a suitable link-level byte size, and I abstract from the fact that there may be fragmentation and reassembly processes performed by intermediate protocols at each end of the link. The details of how the network handles the messages are hidden from the NORM protocol senders and receivers, and therefore it seems natural to abstract from them in the model.

The NORM draft does not explicitly mention *in-order* delivery as part of the service it offers applications, but the protocol is designed to operate successfully even though the network misorders data packets [7]. The fact that it is not mentioned could possibly mean that there can be NORM implementations both with and without an in-order delivery service. Whether a protocol provides in-order delivery or not is not really relevant to the communication between protocol peers, but rather to the communication between a protocol node and its application. Since the NORM specification mentions misordering of messages, I have included in-order delivery as part of my model of the NORM protocol.

4.2 Modelling the Communication Topology

Before we can start working on the specification of what the NORM components do, we need to model a framework within which these actions take place. How are messages sent from one node to another? How are packets multicast? How does packet loss occur? How do we measure time? And how does time affect the system?

The rest of this chapter presents this framework. I present the time domain and define the effect of time elapse on configurations. I also model messages, the sending of packets through links and routers, and define some general properties for network nodes. Finally, I implement a random backoff timeout algorithm [6] used by receivers for suppressing feedback.

The framework I present is based on the corresponding part of Ölveczky's model of the AER/NCA protocol suite [21]. More specifically, I have looked to his model for inspiration on how to model the advance of time in a system, the sending of messages, packet loss in the network and the class hierarchy, but always adapting his ideas to the requirements of my modelling task. His case study was very useful to me in the initial stage of the modelling process. However, the specifications of the RTT and data and repair transmission components presented in subsequent chapters have been developed entirely without using Ölveczky's model as a template, because the NORM protocol is quite different from the AER/NCA protocol.

4.2.1 The Time Domain

To model the time domain, I use the built-in module `NAT-TIME-DOMAIN-WITH-INF`, which defines a time domain of natural numbers, extended with the infinity value `INF`. The `INF` value is useful because we need a way of saying that a timer is turned off. In my model, time advances on an object configuration (i.e., on a configuration which only contains objects) as long as the timer attributes of the objects are running or turned off. The purpose of the timers is to "halt" the advance of time to allow necessary instantaneous actions to take place when they expire.

4.2.2 Timed Object-Oriented Systems

The module `TIMED-OO-SYSTEM` specifies the data type `OidSet` of sets of *object identifiers*, which will be used to denote the set of children of a node.

```
sort OidSet .    subsort Oid < OidSet .
op none : -> OidSet .
op __ : OidSet OidSet -> OidSet [assoc comm id: none] .
```

The module also declares the operators `delta` and `mte` for configurations. `delta` models the *effect* of time advance on a configuration by updating the clock and timer values of objects and messages in the configuration. `mte` computes the *maximal possible time increase* in a configuration, i.e., the greatest amount of time that can pass before some instantaneous action must be taken. Both operators are declared to be *frozen* to make sure that their arguments cannot be rewritten.

```
op delta : Configuration Time -> Configuration [frozen (1)] .
op mte : Configuration -> TimeInf [frozen (1)] .

vars NECF NECF' : NEConfiguration .    var T : Time .

eq delta(none, T) = none .
eq delta(NECF NECF', T) = delta(NECF, T) delta(NECF', T) .
eq mte(none) = INF .
eq mte(NECF NECF') = min(mte(NECF), mte(NECF')) .
```

Initially, I define `delta` and `mte` for configurations only. When I start adding classes of objects and messsages to the model, I will have to give equations for interpreting `delta` and `mte` on *every* component in a configuration.

My specification will only have one tick rule:

```
var OC : NEObjectConfiguration .

crl [tick] :
  {OC} => {delta(OC, mte(OC))} in time mte(OC) if mte(OC) =/= INF .
```

This rule is only used on *object configurations*, i.e., it is only applied when there are no unread "ripe" messages in the configuration—"unripe" messages are travelling along the links and are inside the link objects. The time increase is computed by the function `mte`, which finds the smallest timer value in the configuration. The tick rule advances time to the next moment in time when some instantaneous action can or must be taken. It is fairly easy to see, by inspecting the rewrite rules, that no instantaneous action can happen in the protocol *between* the moments in time that are "visited" by the tick rule. Of course, since the time domain is the natural numbers, another option would be to advance time by `1` in each tick rule application. I choose the former to significantly reduce the number of ("uninteresting") "timestamped states" in later search and model checking analysis. Alternatively, one could also have used a "time-nondeterministic" tick rule [26], together with appropriate *time sampling strategies.*

4.2.3 Commonly Used Variables

The modules that will be described in the rest of this chapter have some of their variable declarations in common. To avoid unnecessary repetition, I list the variable declarations of the modules below.

```
vars N N' N'' N''' : Nat .
vars NZN NZN' NZN'' NZN''' : NzNat .
vars R R' : Rat .
vars T T' : Time .
var NZT : NzTime .
var M : Msg .
vars ML ML' : MsgList .
var CP : ControlPacket .
var DP : DataPacket .
vars DUI DUI' : DataUnitId .
vars DUIL DUIL' : DataUnitIdList .
vars O O' R L : Oid .
var OS : OidSet .
```

4.2.4 Messages

The module `MESSAGES` defines the messages used by the NORM sender and receivers, and some functions related to these.

Objects store their messages is lists. New messages are entered from the right, so the leftmost message is the oldest one in the list:

```
sort MsgList .    subsort Msg < MsgList .
op nil : -> MsgList [ctor] .
op _++_ : MsgList MsgList -> MsgList [ctor assoc id: nil] .
```

A message travelling through a link or stored in the buffer of a router has a delay value attached to it, denoting the time left before the message will leave the object:

```
msg dly : Msg Time -> Msg .
```

By extending the operator `delta` so that it decreases the attached delay value whenever time passes in the configuration, we model the effect of time on messages travelling through the network:

```
op delta : MsgList Time -> MsgList [frozen (1)] .
op delta : Msg Time -> Msg [frozen (1)] .

eq delta((nil).MsgList, T) = (nil).MsgList .
ceq delta(ML ++ ML', T) =
   delta(ML, T) ++ delta(ML', T) if ML =/= nil /\ ML' =/= nil .
eq delta(dly(M, T), T') = dly(M, T monus T') .
```

The leftmost (oldest) message in a list will have the least delay and the rightmost (newest) the greatest:

```
op leastDly : MsgList -> TimeInf .
op greatestDly : MsgList -> TimeInf .

eq leastDly(nil) = INF .
eq leastDly(dly(M, T) ++ ML) = T .

eq greatestDly(nil) = 0 .
eq greatestDly(ML ++ dly(M, T)) = T .
```

Identification of Data Content

In Section 3.2.2, I explained how the segments of an object are identified by an object number and a segment number. In my model of NORM, a pair of such identifiers (a transport data unit identifier) is a term of sort `DataUnitId` :

```
sort DataUnitId .
op _::_ : Nat Nat -> DataUnitId .
```

In general, the data unit ids will have the form $m :: n$, where m and n are non-zero natural numbers. These data unit ids identify one segment specifically. However, sometimes we will see data unit ids of the form $m :: 0$. These are used by a receiver when sending a repair request for an entire missing object. In addition, the data unit identifier $0 :: 0$ will be used in the data transmission component to indicate that a sender or receiver has not yet begun sending or receiving data, or that transmission or reception is completed. Lists of such identifiers are defined as follows:

```
sort DataUnitIdList .    subsort DataUnitId < DataUnitIdList .
op nil : -> DataUnitIdList [ctor] .
op _;_ : DataUnitIdList DataUnitIdList
         -> DataUnitIdList [ctor assoc id: nil] .
```

NORM Messages

A NORM message is either an atomic control message of sort `ControlPacket`, used to perform certain protocol functions, or a data carrying message of sort `DataPacket`.

```
sorts ControlPacket DataPacket .
subsort ControlPacket < Msg .    subsort DataPacket < Msg .
```

The reason for having two sorts of messages in the model is that different protocol messages have different sizes. In Section 4.2.5, when I model the delay of messages travelling through a link, the data packet and the control packets will be given a size of 1500 and 64 bytes, respectively. The NORM draft operates with a number of different packet sizes, some fixed, some variable. I have adopted Ölveczky's definition of packet sizes in [21], trusting that these values—suggested by the developers of the AER/NCA protocol—represent common sizes for packets.

Every NORM message contains information that is used by the different components. Since I will only be modelling the data transmission and RTT components, the messages of the model will only include the information necessary for performing the tasks in those components. The `MESSAGES` module declares the following messages: `DATA`, `FLUSH`, `SQUELCH`, `CC`, `NACK` and `ACK`. Their definitions and use will be given in Chapters 5 and 7.

4.2.5 The Network

This section defines the network topology and multicast transmission between the nodes. It also describe how packet loss occurs in the model.

The Class Hierarchy and General Properties of Nodes

The module `NODES` declares a number of classes, each representing one or more properties of a network node:

```
class RootOrLeaf | clock : Time, NormRobustFactor : NzNat,
                   GRTT : Time, gsize : NzNat .
class Parent | children : OidSet  .
class Child | parent : Oid .
class RandomSeed | randomSeed : Nat .
class Sender | sendRate : Time .
class Receiver | CLR : Bool .

subclass Sender < Parent RootOrLeaf .
subclass Receiver < RootOrLeaf Child RandomSeed .
```

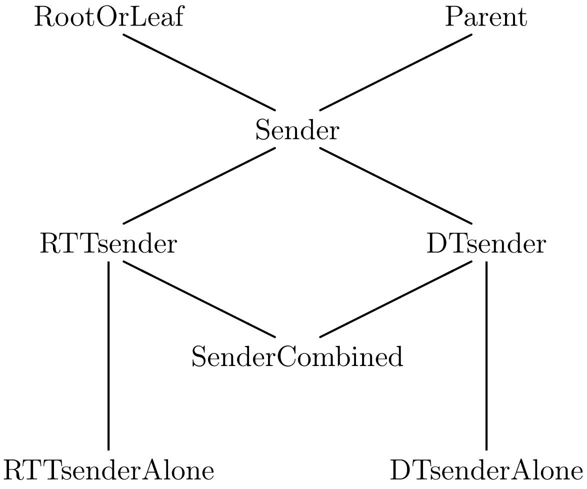

Figure 4.2: The sender class hierarchy

The `Sender` and `Receiver` classes define the basic properties of every sender and receiver in the model. Both classes have the following four basic attributes in common: `clock` (the object's internal clock), `NormRobustFactor` (a fixed value used in setting the timers), `GRTT` (the group's greatest round-trip time) and `gsize` (the size of the receiver group). In addition, a sender has the basic attributes `children` (the set of nodes that are directly reachable from the sender) and `sendRate`, and a receiver has `parent` (the node above it in the multicast tree), `randomSeed` and `CLR` (the current limiting receiver). The `sendRate` attribute gives the (fixed) transmission interval in milliseconds between each data content message, and is used to set the sender's transmission timer. The receivers use the `randomSeed` value in computing their random backoff timeouts.

For the specification of the RTT and data transmission components, I will declare the subclasses `RTTsender`, `RTTreceiver`, `DTsender` and `DTreceiver` with the necessary attributes for performing the functions of these subprotocols, and declare classes `SenderCombined` and `ReceiverCombined` for combined execution of the specifications. The resulting class hierarchy for the sender is shown in Figure 4.2—the class hierarchy for the receiver is almost identical.

When modelling transitions as rewrite rules, one has to consider whether a transition involves actions in both the RTT and data transmission components simultaneously, or whether only one of them is affected. For example, there might be a transition where a sender reads a message and performs one or more actions. If those actions only affect the RTT component, we specify a rule for objects of the class `RTTsender`, which is used both in stand-alone execution of the RTT component and in executing the combined protocol. On the other hand, if the sender performs actions affecting both the RTT *and* the data transmission component, we have to specify rules for stand-alone execution of these components, and in addition specify the transition for the class `SenderCombined` for the combined protocol.

Links

When a network node wishes to send a message, it puts the message in a message "wrapper" before sending it to the link(s) that connects it to its neighbour(s). A link uses a similar wrapper when delivering a message to a node. In general, messages are released into the configuration where they are picked up by the next link or node in the path to their destination. The message wrappers identify the previous stop on the path and the next one. Note that releasing a message into the configuration and picking it up takes zero time, because the tick rule cannot be applied as long as there are messages in the configuration. Thus, adding such message wrappers does not change the time properties of the model.

```
msg intoLink_from_to_ : Msg Oid Oid -> Msg .
msg outOfLink_from_to_ : Msg Oid Oid -> Msg .
```

A link connects a pair of nodes and supports data flowing in both directions simultaneously. The values of the link attributes `upNode` and `downNode` identify the pair of nodes the link connects, and `upstream` and `downstream` are the message streams between the two. The time it takes for a message to cross a link is in part a function of the link's *propagation delay* and its *bandwidth*. Propagation delay is the speed-of-light limitation on the time it takes to send a signal across a physical medium, and bandwidth (or link speed) is a measure of the number of signals that can be transmitted across a link per second [28]. The propagation delay and bandwidth of a link is given in the initial state of a system, where propagation delay is expressed in milliseconds and bandwidth in Mbps (megabits per second).

```
class Link |
  upNode : Oid, downNode : Oid,
  upstream : MsgList, downstream : MsgList,
  propDelay : NzTime, bandwidth : NzNat .
```

The function `transDelay` calculates the *transmission delay* of data and control packets. The transmission delay of a packet in a link is the packet size divided by the bandwidth. `transDelay` takes two arguments, a packet and the bandwidth of a link in Mbps. Packets of sort `ControlPacket` are 64 bytes large ($64 * 8$ bits), and packets of sort `DataPacket` are 1500 bytes large ($1500 * 8$ bits).

```
op transDelay : Msg NzNat -> Time .

*** The transmission delay of control packets of 64 bytes:
eq transDelay(CP, NZN) =
   ((64 * 8) + ((NZN * 1000) monus 1)) quo (NZN * 1000) .

*** The transmission delay of data packets of 1500 bytes:
eq transDelay(DP, NZN) =
   ((1500 * 8) + ((NZN * 1000) monus 1)) quo (NZN * 1000) .
```

The following rule models how a message is entered into a link—in this case to be transported *downstream* in the multicast tree—and the delay of the message is computed. The delay (or latency) of the message is calculated by adding the link's propagation delay to the message's transmission delay. However, if the link's propagation delay is less than the greatest delay value of the message stream, this last value is added to the message transmission delay instead.

```
crl [msgFromUpNode] :
  (intoLink M from O to O')
  < L : Link |
    upNode : O, downNode : O', downstream : ML,
    propDelay : NZT, bandwidth : NZN >
  =>
  < L : Link |
    downstream : ML ++
                 dly(M, max(NZT, greatestDly(ML)) + transDelay(M, NZN)) >
  if leastDly(ML) =/= 0 .
```

When the message has crossed the link, it is released into the configuration to be picked up by the recipient:

```
rl [msgToDownNode] :
  < L : Link |
    downNode : O, upNode : O', downstream : dly(M, 0) ++ ML >
  =>
  < L : Link | downstream : ML >
  (outOfLink M from O' to O) .
```

Two similar rules, `[msgFromDownNode]` and `[msgToUpNode]`, are also defined for messages going *upstream*. These rules can be found in Appendix B.

Finally, we add equations for `delta` and `mte`, which, respectively, decreases the delay values of the messages in a link, and finds the least delay value among the messages:

```
eq delta(< L : Link | downstream : ML, upstream : ML' >, T) =
  < L : Link | downstream : delta(ML, T), upstream : delta(ML', T) > .

eq mte(< L : Link | downstream : ML, upstream : ML' >) =
  min(leastDly(ML), leastDly(ML')) .
```

Routers

The routers form the "trunk" of the multicast tree, connecting different nodes via the links. Router objects store the incoming messages in a `buffer`, which has a certain capacity. The maximum buffer capacity of a router is the value of `bufferCap` plus the number of messages in `buffer`. `queuingDelay` states how long it takes before the router can forward a message.

```
class Router |
  buffer : MsgList, bufferCap : Nat, queuingDelay : Time .

*** Routers have a parent and a set of children:
subclass Router < Parent  Child .
```

An incoming packet is stored with information about who sent it to the router. This is done to make sure that the packet is not sent back to the sender when the router forwards it.

```
msg forward : Oid Msg -> Msg .   *** forward Msg from Oid.
```

This is certainly a very crude model of a router, but it will suffice for modelling packet loss in a network. I could have specified the router such that the value of queuingDelay is computed as a function of some fixed delay value, which shows how long a message is kept in an otherwise empty buffer before it is forwarded, and the greatest delay of the buffer. However, it would not have made the model more sophisticated, as a very low buffer capacity is required to cause packet loss.

As the next rule shows, an incoming packet is buffered as long as the router has available buffer space. If its capacity is zero, it will simply drop the packet.

```
rl [bufferOrDrop] :
  (outOfLink M from O to R)
  < R : Router |
    buffer : ML, bufferCap : N, queuingDelay : N' >
  =>
  if N > 0 then
  < R : Router |
    buffer : ML ++ dly(forward(O, M), N'), bufferCap : sd(N, 1) >
   else < R : Router | > fi .
```

When the router is ready to send a packet, it forwards it to all connections except the one where the message came from.

```
rl [forward] :
< R : Router |
  parent : O, children : OS,
  buffer : dly(forward(O', M), 0) ++ ML, bufferCap : N >
=>
< R : Router | buffer : ML, bufferCap : N + 1 >
if O' == O
 then (multisend M from R to OS)
 else (multisend M from R to O (OS ex O')) fi .
```

multisend_from_to_ takes a message and a set of object identifiers—the receivers—and creates a link "wrapper" for each message:

```
msg multisend_from_to_ : Msg Oid OidSet -> Configuration .

eq multisend M from O to none = none .
eq multisend M from O to (O' OS) =
   (intoLink M from O to O') (multisend M from O to (OS ex O')) .


*** Pick out all occurences of an Oid from a set of Oids.
op _ex_ : OidSet Oid -> OidSet .

eq none ex O = none .
eq (O OS) ex O' = if O == O' then (OS ex O') else O (OS ex O') fi .
eq O O = O .
```

As with the link class, delta decreases the delay of packets in the router's buffer, and mte returns the smallest delay value among the messages:

```
eq delta(< R : Router | buffer : ML >, T) =
   < R : Router | buffer : delta(ML, T) > .
eq mte(< R : Router | buffer : ML >) = leastDly(ML) .
```

Alternative Ways of Modelling Packet Loss

In this model I have chosen to let packet loss occur when there is too much traffic going in to the routers. It seemed like an appropriate way of modelling packet loss, since it reflects a common situation in the Internet, where congestion of the network causes routers and switches to drop packets

This is not the only way to model packet loss. A different solution was chosen by Ölveczky in [21]: the links are modelled with an upper bound on the number of packets they can hold (regardless of the packet sizes). The capacity decreases as new packets enter the link, and when a node tries to send a packet through a link that is full, the packet is dropped.

The router defined above has a buffer with an upper bound on the number of packets it can hold. Another possibility is to make the router more abstract by letting it drop packets at random.

4.2.6 Setting the Receiver Backoff Timeout Values

This section presents the NORM protocol algorithm for suppressing receiver feedback, and my implementation of it.

In order to keep the amount of feedback traffic to the sender as low as possible, the NORM protocol uses various timeout mechanisms for suppressing the receiver feedback messages. When a receiver discovers that one or more data segments are missing, it waits a while before sending a repair request. The algorithm for calculating such *backoff timeouts* is described in Section 3.2.2 of the NORM building block document [6]. The algorithm generates random backoff timeout values with a truncated exponential distribution. In addition to suppression of `NACK` messages, the algorithm is also part of the RTT feedback suppression function, which will be further described in Chaper 5.

Based on

- the receiver group size (R), and
- a maximum allowed backoff timeout value ($T\_maxBackoff$) of $K * GRTT$, where K is a constant factor (the NORM robust factor) and $GRTT$ is the group's greatest round-trip time,

the algorithm calculates random backoff timeouts as follows:

1. Establish an optimal mean L for the exponential backoff based on the group size:

$$L = \ln(R) + 1$$

2. Pick a random number x from a uniform distribution over a range of:

$$\frac{L}{T\_maxBackoff * (\exp(L) - 1)} \quad to \quad \frac{L}{T\_maxBackoff * (\exp(L) - 1)} \quad + \quad \frac{L}{T\_maxBackoff}$$

3. Use the random number x and the optimal mean L to generate a timeout value t' with the equation

$$t' = T\_maxBackoff / L * \ln(x * (\exp(L) - 1) * (T\_maxBackoff / L))$$

The rest of this section contains my Real-Time Maude implementation of the random backoff timeout algorithm. The maximum backoff window, $T\_maxBackoff$, is implemented by the function `maxBackoff`:

```
*** maxBackoff(NormRobustFactor, GRTT)
op maxBackoff : NzNat Nat -> Nat .
eq maxBackoff(NZN, N) = NZN * N .
```

Step 1 of the algorithm is implemented by the function `lambda`:

```
*** lambda(groupSize)
op lambda : Rat -> Rat .
eq lambda(R) = log(R) + 1 .
```

The function x below corresponds to step 2 of the algorithm, and provides the random number x:

```
*** op x(seed, NormRobustFactor, GRTT, groupSize)
op x : Nat NzNat Nat NzNat -> Rat .

eq x(N, NZN, N', NZN') =
  randomUpTo(N, lambda(NZN') / maxBackoff(NZN, N'))
  + lambda(NZN') / (maxBackoff(NZN, N') * (exp(lambda(NZN')) - 1)) .
```

The function x uses the function randomUpTo, which returns a random value between 0 and 1:

```
*** Find a random value between 0-1:
*** randomUpTo(seed, upper)
op randomUpTo : Nat Rat -> Rat .
eq randomUpTo(N, R) = randomFrac(N) * R .

*** randomFrac(seed)
*** Returns a fraction value from 0 to 1.
op randomFrac : Nat -> Rat .
eq randomFrac(N) = random(N) rem (MAX-VALUE + 1) / MAX-VALUE .

op MAX-VALUE : -> Nat .
eq MAX-VALUE = 14 .
```

In the implementation of the algorithm, I have used a random function from Ölveczky's specification of the AER/NCA protocol, which generates a pseudo-random sequence of natural numbers [21]. The function is used by randomFrac:

```
op random : Nat -> Nat .
eq random(N) = ((104 * N) + 7921) rem 10609 .
```

Finally, step 3 of the random backoff timeout algorithm is implemented by the function randomBackoff, which generates the timeout value t':

```
*** randomBackoff(seed, NormRobustFactor, GRTT, groupSize)
op randomBackoff : Nat NzNat Nat NzNat -> Nat .

eq randomBackoff(N, NZN, N', NZN') =
  nat((maxBackoff(NZN, N’) / lambda(NZN’))
       * log(x(N, NZN, N’, NZN’)
             * (exp(lambda(NZN’)) - 1)
             * (maxBackoff(NZN, N’) / lambda(NZN’)))) .
```

The definition of the functions `log`, `exp` and `nat` can be found in Appendix B.

Chapter 5

Specifying the Round-Trip Time Component

This chapter specifies the round-trip time component formally. The function and actions of the component can be summarised as follows:

- **Function:** To provide a common timeout basis for the senders and receivers.
- **Actions:**
  - The sender collects RTT values from the receivers, and uses these values to estimate the group's greatest round-trip time (GRTT), and
  - the sender announces the GRTT in every message it multicasts to the receivers.

After a brief discussion of some aspects of the informal specification, and of some decisions made prior to the modelling, the rest of the chapter presents the Real-Time Maude specification of the RTT component. The formal specification is based on Sections 5.5.1 and 5.5.2 of the NORM specification and Section 3.7 of the building block document.

5.1 Comments on the Informal Specification

The GRTT measurement and announcement procedures are carefully described in the informal NORM specification and the building block document, and modelling the procedures as Real-Time Maude rewrite rules is fairly straightforward. There is, however, one detail in the feedback suppression procedure that is worth commenting on. The NORM specification says that a receiver shall cancel its pending NORM_ACK feedback message if 1) it is about to provide sender feedback with a NORM_NACK message, or 2) it receives a NORM_ACK from another member of the group whose data reception rate is "sufficiently close to or less than" its own[1] [7]. The second condition ensures that the receiver with the lowest rate reports.

[1] Or, in cases where the receivers unicast their feedback, a NORM_CMD(REPAIR_AVD) that announces such a rate (see Appendix A).

The NORM_NACK message also carries this information, and it seems logical to exploit this for RTT feedback suppression purposes: a receiver should cancel its feedback message if it receives a NORM_ACK *or* a NORM_NACK from another receiver with a smaller reception rate. The version of the NORM protocol that I model in this thesis does not take advantage of this information, however, in the current version of the protocol specification, the information in the NORM_NACK message is used in suppressing RTT feedback [5].

5.2 Comments on the Real-Time Maude Specification

Provided that the sender has received some feedback from the group, it's NORM_CMD(CC) messages will contain information about the congestion control state of one or more receivers, such as their RTT values and which receiver is the current limiting receiver. In our model, the sender does not announce such information, for the following reasons: Firstly, since I have already decided not to model the congestion control part of NORM, it seems reasonable to abstract away any information that does not directly concern GRTT measurement. The receivers do not require knowledge of their own RTT to be able to provide GRTT feedback to the sender. Secondly, the current limiting receiver is identified in an initial state for the system. This means that there is no need to inform that receiver of its status—it already knows and provides feedback accordingly. However, the Real-Time Maude specification can easily be extended to include this feature.

5.3 The `RTTsender` and `RTTreceiver` Classes

The timed object-oriented module `NORM-RTT` contains class declarations for the stand-alone RTT component and for the combined protocol.

```
class RTTsender |
  sendRateInKbps : Nat,
  CCtransTimer : TimeInf,
  peakRTT : Time,
  CLRresponse : Bool,
  lowPeakRTTcounter : Nat .
```

A `RTTsender` records the highest RTT values it receives in `peakRTT`, and sets `CLRresponse` to true if the RTT came from the current limiting receiver. It keeps track of the occurence of low peak RTT values in `lowPeakRTTcounter` and uses `CCtransTimer` to trigger the retransmission of the NORM_CMD(CC) message. Finally, the `sendRateInKbps` attribute contains its send rate measured in bytes per second.

```
class RTTreceiver |
  ACKtimer : TimeInf,
  ACKholdoffTimer : TimeInf,
  timestamp : Time,
  receivedTimestamp : Time,
  rcvRateInKbps : Nat,
  sndRateInKbps : Nat .
```

A `RTTreceiver` stores the timestamp it receives from a sender in the attribute `timestamp` and the time it received it in `receivedTimestamp`. When `ACKtimer` expires, the receiver returns the adjusted timestamp, and afterwards it does not respond to NORM_CMD(CC) messages as long as `ACKholdoffTimer` is running. The receiver records the senders transmission rate (`sndRateInKbps`) and its own rate of reception (`rcvRateInKbps`). The `CLR` attribute is set to true if the receiver is the current limiting receiver.

Both classes inherit the following attributes from their common superclasses: `clock`, `GRTT`, `gsize`, and `NormRobustFactor`. In addition, `RTTsender` inherits the attributes `children` and `sendRate` from its superclass `Sender`, and `RTTreceiver` inherits `randomSeed`, `parent` and `CLR` from `Receiver`.

```
  subclass RTTsender < Sender .
  subclass RTTreceiver < Receiver .
```

Finally, I declare subclasses for specification and execution of the rules of the stand-alone RTT component. The receivers can provide round-trip time feedback not only in NORM_ACK messages, but also in NORM_NACK messages. The NACK message belongs to a different component, but in order to make the stand-alone specification correct, the receivers has to be able to send NACK messages. Therefore, the class `RTTreceiverAlone` has an attribute `nacksToBeSent` that is used to simulate transmission of repair requests.

```
  class RTTsenderAlone .
  subclass RTTsenderAlone < RTTsender .

  class RTTreceiverAlone | nacksToBeSent : MsgList .
  subclass RTTreceiverAlone < RTTreceiver .
```

5.4 Commonly Used Variables

The following variables are used in the rules and equations of the RTT component:

```
  vars S R O O' : Oid .
  var OS : OidSet .
  var M : Msg .
  vars ML ML' : MsgList .
  var DUIL : DataUnitIdList .
  vars N N' N'' N''' N'''' : Nat .
```

```
vars NZN NZN' : NzNat .
vars T T' T'' T''' T'''' : Time .
var NZT : NzTime .
vars TI TI' : TimeInf .
vars B B' : Bool .
```

5.5 Messages

The RTT model uses some of the messages defined in the module MESSAGES, which was described in Section 4.2.4.

In NORM, the NORM_CMD(CC) message is used both in congestion control and in GRTT measurement. Since I will not be modelling the congestion control part of the protocol, I have only included information that is relevant to GRTT measurement in the CC message.

The sender uses the CC message to send a timestamp to the receivers. As in every NORM sender message, the current GRTT estimate is included. In addition, the sender advertises its current data transmission rate. In the NORM specification, the message also contains the size of the receiver group. However, since the number of participants will not change during a run, I have chosen to let the group size be known to each member in the initial state (as an attribute). The receivers use the GRTT estimate and the group size to set their timestamp feedback timeouts.

```
*** Usage: CC(timestamp, grtt, sendRateInKbps)
msg CC : Time Time Nat -> ControlPacket .
```

In the RTT component, the receivers primarily use ACK messages to return an adjusted version of the sender's timestamp. When the CLR flag is set to true, the sender will know that this is the RTT for the current limiting receiver, and will use it to set its CC transmission timer. The rate at which data is received is included because it is used by the receivers in suppressing their feedback to the sender.

```
*** Usage: ACK(adjustedTimestamp, rcvRateInKbps, CLRflag)
msg ACK : Time Nat Bool -> ControlPacket .
```

The NACK message also returns an adjusted timestamp, and is used for RTT feedback when the receiver has pending repair needs. It does not include the receive rate, for reasons explained in Section 5.7. The use of NACK messages in the repair request procedure will be explained in Chapter 7.

```
*** Usage: NACK(dataUnitIdList, adjustedTimestamp, CLRflag)
msg NACK : DataUnitIdList Time Bool -> SmallPacket .
```

Finally, a constant NACK is defined for use in the stand-alone protocol to simulate a receiver's pending repair need:

```
  msg NACK : -> ControlPacket .
```

5.6 The RTTsender Rules

The four sender rules in this section describe actions that are explicitly stated in the NORM specification.

5.6.1 Initialize RTT collection

The sender initializes RTT collection by transmitting a series of NORM_CMD(CC) messages with a GRTT of 500 ms until it receives some feedback from the receivers. This initial GRTT is also used as the NORM_CMD(CC) transmission timer value. Before advertising the GRTT the sender checks that it is at least as great as its current data transmission rate (sendRate)—if not, this last value is advertised instead.

```
  crl [initializeRTTcollection] :
    < S : RTTsender |
      children : OS, clock : T, sendRateInKbps : N, GRTT : 500,
      sendRate : T', CCtransTimer : TI, peakRTT : 0 >
    =>
    < S : RTTsender |
      CCtransTimer : 500 >
    (multisend CC(T, max(500, T'), N) from S to OS)
    if TI == INF or TI == 0 .
```

5.6.2 Receive Adjusted Timestamp

When a receiver provides feedback, either in an ACK or a NACK message, it has adjusted the timestamp included by adding the amount of time it held the timestamp. Thus, the current time minus the timestamp reflects the actual time the message spent in the network. The sender finds the receiver's RTT by subtracting the adjusted timestamp from the current value of clock. The sender updates the GRTT if the received RTT is larger than the GRTT and the RTT it has received previously. Otherwise, it simply stores the new RTT, provided it is greater than the old RTT. To make the two rules below easier to read, I use conditional rewrite rules with matching equations in their conditions [14].

```
  var RECEIVED-RTT : Time .
```

```
*** Receives explicit feedback, i.e., an ACK message.
crl [receiveAdjustedTimestamp1] :
  (outOfLink ACK(T, N, B) from O to S)
  < S : RTTsender |
    clock : T', GRTT : T''', peakRTT : T'' >
  =>
  (if RECEIVED-RTT > T''' and RECEIVED-RTT > T''
  then
   *** update GRTT
  < S : RTTsender |
    peakRTT : RECEIVED-RTT, CLRresponse : B,
    GRTT : updateGRTT(T''', RECEIVED-RTT) >
  else
   *** keep new peak if larger than old peak
  < S : RTTsender |
    peakRTT : (if RECEIVED-RTT > T'' then RECEIVED-RTT else T'' fi),
    CLRresponse : B > fi)
if RECEIVED-RTT := (T' monus T) .


*** Receives implicit feedback, i.e., a NACK message.
crl [receiveAdjustedTimestamp2] :
  (outOfLink NACK(DUIL, T, B) from O to S)
  < S : RTTsenderAlone |
    clock : T', GRTT : T''', peakRTT : T'' >
  =>
  (if RECEIVED-RTT > T''' and RECEIVED-RTT > T''
  then
   *** update GRTT
  < S : RTTsenderAlone |
    peakRTT : RECEIVED-RTT, CLRresponse : B,
    GRTT : updateGRTT(T''', RECEIVED-RTT) >
  else
   *** keep new peak if larger than old peak
  < S : RTTsenderAlone |
    peakRTT : (if RECEIVED-RTT > T'' then RECEIVED-RTT else T'' fi),
    CLRresponse : B > fi)
if RECEIVED-RTT := (T' monus T) .
```

5.6.3 Update the GRTT Estimate at the End of the Collection Period

At the end of a RTT collection period, when the CCtransTimer expires, the GRTT estimate is updated only if the current peakRTT is less than the current GRTT and this is the third collection period in a row with a peak RTT value smaller than the GRTT (i.e., the lowPeakRTTcounter is 2). If this is not the case then the GRTT has already been updated during the RTT collection period. If the RTT has been used for updating, then peakRTT is set to 0, otherwise the sender keeps the current value. A CC is multicast to the receivers, containing the current GRTT estimate. However, if the sender's transmission rate is greater than estimate, this rate is advertized as the current GRTT instead.

The CCtransTimer is reset either with the current GRTT value, or with the peakRTT value, if this was provided by the CLR. However, if the sendRate value is greater than the GRTT or the current limiting receiver's RTT, the timer is reset to this value.

```
  crl [endOfRTTcollectionPeriod] :
    < S : RTTsender |
      children : OS, clock : T, GRTT : T'',
      sendRate : T''', sendRateInKbps : N', CCtransTimer : 0,
      peakRTT : T', CLRresponse : B, lowPeakRTTcounter : N >
    =>
    (if B == true and T''' <= T'
    then
    (< S : RTTsender |
       CCtransTimer : T',
       peakRTT : (if (T' >= T'') then 0 else T' fi),
       lowPeakRTTcounter : (if ((T' >= T'') or (N == 2))
                            then 0 else (N + 1) fi),
       GRTT : (if ((T' < T'') and (N == 2)) then updateGRTT(T'', T')
              else T'' fi) >
    *** The CC contains the most recent GRTT estimate
 (multisend CC(T, max((if ((T' < T'') and (N == 2)) then updateGRTT(T'', T')
               else T'' fi) , T'''), N') from S to OS))
     else
    (< S : RTTsender |
       CCtransTimer : max((if ((T' < T'') and (N == 2)) then
         updateGRTT(T'', T') else T'' fi), T'''),
       peakRTT : (if (T' >= T'') then 0 else T' fi),
       lowPeakRTTcounter : (if ((T' >= T'') or (N == 2))
                            then 0 else (N + 1) fi),
       GRTT : (if ((T' < T'') and (N == 2)) then updateGRTT(T'', T')
              else T'' fi) >
 (multisend CC(T,  max((if ((T' < T'') and (N == 2)) then updateGRTT(T'', T')
               else T'' fi), T'''), N') from S to OS)) fi)
    if T' =/= 0 or (T' == 0 and T'' < 500) .
```

5.6.4 The updateGRTT Function

The sender uses the following function to update its GRTT estimate, where peak is the peak RTT value recorded during a probing period, and current_estimate is the sender's current estimate of the GRTT [6]:

```
      if (peak > current_estimate)
```

```
        current_estimate = 0.25 * current_estimate + 0.75 * peak;
    else
        current_estimate = 0.75 * current_estimate + 0.25 * peak;
```

The function is implemented by the updateGRTT operator:

```
*** updateGRTT(GRTTestimate, peakRTT)
op updateGRTT : Nat Nat -> Time .
eq updateGRTT(N, N') =
  nat(if N' > N then 1/4 * N + 3/4 * N'
                else 3/4 * N + 1/4 * N' fi) .
```

5.7 The RTTreceiver Rules

Unlike the sender rules in Section 5.6, not all the receiver rules model actions that are explicit in the NORM specification, and I will mention which these are as I explain the rules.

5.7.1 Current Limiting Receiver Feedback

The current limiting receiver (CLR) will respond immediately to a CC from the sender. Both its timers are off from the start and the CLR does not have a holdoff timeout after acknowledging the RTT request.

```
rl [CLRfeedback] :
  (outOfLink CC(T, T', N) from O to R)
  < R : RTTreceiver |
    parent : O, clock : T'', GRTT : T''', CLR : true, rcvRateInKbps : N' >
  =>
  < R : RTTreceiver |
    GRTT : T', sndRateInKbps : N, timestamp : T, receivedTimestamp : T'' >
  (intoLink ACK(T, N', true) from R to O) .
```

5.7.2 Initialize an ACK Feedback Cycle

When a non-CLR receives a CC, it stores the timestamp and sets the ACKtimer provided both this and the ACKholdoffTimer are turned off.

```
rl [initializeACKcycle] :
  (outOfLink CC(T, T'', N) from O to R)
  < R : RTTreceiver |
    clock : T', gsize : NZN, randomSeed : N',
    NormRobustFactor : NZN', rcvRateInKbps : N'',
    ACKtimer : INF, ACKholdoffTimer : INF, CLR : false >
  =>
  < R : RTTreceiver |
    timestamp : T, receivedTimestamp : T', GRTT : T'',
    randomSeed : random(N'), sndRateInKbps : N,
    ACKtimer : RTTbackoff(random(N'), NZN', T'', NZN, N'', N) > .
```

The next rule models how the receiver ignores further CC messages from the sender during the feedback timeout, i.e., when the ACKtimer is set. This is not stated explicitly in the specification. The variable NZT of sort NzTime in the ACKtimer means that the timer is *running*.

```
  rl [receivesCCduringBackoffTimeout] :
    (outOfLink CC(T, T', N) from O to R)
    < R : RTTreceiver | ACKtimer : NZT >
    =>
    < R : RTTreceiver | GRTT : T' > .
```

5.7.3 Cancel ACK Feedback Message

During the feedback timeout, a receiver will cancel a pending ACK if it is about to send another feedback message, i.e., a NACK message, as shown in the next rule. The receiver includes the adjusted timestamp in the NACK and sets its ACKholdoffTimer. This rule ensures that a pending NACK is always sent, even when the receiver is not in a RTT feedback cycle (i.e., the ACKholdoffTimer may be running) or if it is the CLR.

```
  rl [cancelACK1] :
    < R : RTTreceiverAlone |
      parent : O, clock : T, timestamp : T', receivedTimestamp : T'',
      ACKtimer : TI, ACKholdoffTimer : TI',
      NormRobustFactor : NZN, GRTT : N,
      nacksToBeSent : dly(NACK, 0) :: ML, CLR : B >
    =>
    < R : RTTreceiverAlone |
      ACKtimer : (if TI =/= INF and TI =/= 0 then INF else TI fi),
      ACKholdoffTimer : (if TI =/= INF and TI =/= 0
                         then NZN * N else INF fi),
      nacksToBeSent : ML >
    (intoLink NACK(nil, (if T' == 0 then 0 else T' + (T monus T'') fi), B)
    from R to O) .
```

A receiver may also cancel its feedback if it receives an ACK from another receiver who has a data reception rate close to or less than its own, i.e., when the receiver's rate is greater than 0.9 times the rate from the other receiver.

```
rl [cancelACK2] :
  (outOfLink ACK(T, N, B) from O to R)
  < R : RTTreceiver |
    rcvRateInKbps : N', ACKtimer : NZT, ACKholdoffTimer : INF,
    NormRobustFactor : NZN, GRTT : NZN' >
  =>
  (if N' > (N monus (N quo 10))
  *** cancelled if the competing rate is sufficiently close to or less
  *** than N', i.e. N' > N * 0,9
  then
  < R : RTTreceiver | ACKtimer : INF, ACKholdoffTimer : NZN * NZN' >
  else
  < R : RTTreceiver | > fi) .
```

As we saw in Section 5.1, the receiver does not cancel its pending ACK even though it receives a NACK message with a smaller data reception rate than its own. The next rule states that a receiver whose ACKtimer is running will ignore NACK messages from other receivers.

```
rl [ignoreNACK2] :
  (outOfLink NACK(DUIL, T, B) from O to R)
  < R : RTTreceiverAlone |
    ACKtimer : NZT, ACKholdoffTimer : INF >
  =>
  < R : RTTreceiverAlone | > .
```

5.7.4 Send Adjusted Timestamp

When the ACKtimer expires, the receiver multicasts its ACK with the adjusted timestamp and sets its holdoff timer. The receiver adjusts the sender's timestamp by adding the time it held it before providing feedback.

```
rl [sendAjustedTimestamp] :
  < R : RTTreceiver |
    parent : O, clock : T, timestamp : T', receivedTimestamp : T'',
    rcvRateInKbps : N, NormRobustFactor : NZN, GRTT : NZN',
    ACKtimer : 0, ACKholdoffTimer : INF >
  =>
  < R : RTTreceiver |
    ACKtimer : INF, ACKholdoffTimer : NZN * NZN' >
  (intoLink ACK(T' + (T monus T''), N, false) from R to O) .
```

5.7.5 Holdoff Timeout

During the holdoff timeout, the receiver ignores all further CCs from the sender.

```
rl [receivesCCduringHoldoff] :
  (outOfLink CC(T, T', N) from O to R)
  < R : RTTreceiver | ACKtimer : INF, ACKholdoffTimer : NZT >
  =>
  < R : RTTreceiver | GRTT : T' > .
```

ACK and NACK messages from the environment are not useful to a receiver whose holdoff timer is running or who is not in a feedback cycle. The CLR always ignores messages from other receivers. The next two rules model both these cases, which are implicit in the specification.

```
crl [ignoreACK] :
  (outOfLink ACK(T, N, B) from O to R)
  < R : RTTreceiver |
    ACKtimer : INF, ACKholdoffTimer : TI, CLR : B' >
  =>
  < R : RTTreceiver | >
  if (TI =/= 0) or (TI == INF) or (B' == true) .

crl [ignoreNACK1] :
  (outOfLink NACK(DUIL, T, B) from O to R)
  < R : RTTreceiverAlone |
    ACKtimer : INF, ACKholdoffTimer : TI, CLR : B' >
  =>
  < R : RTTreceiverAlone | >
  if (TI =/= 0) or (TI == INF) or (B' == true) .
```

The last rule simply turns off the ACKholdoffTimer when it has expired, so that the receiver can start a new feedback cycle should a new CC arrive.

```
rl [turnOffACKholdoffTimer] :
  < R : RTTreceiver | ACKholdoffTimer : 0 >
  =>
  < R : RTTreceiver | ACKholdoffTimer : INF > .
```

5.7.6 Receiver Feedback Timeout Function

Section 5.5.2.2 of the NORM specification describes the algorithm used by receivers to determine how long they should wait before generating an ACK message. The timeout is calculated as

$$y * r * (K * \mathit{GRTT}) + (1 - y) * \mathit{RandomBackoff}(K * \mathit{GRTT}, GSIZE)$$

where K is the Norm robust factor, GRTT is the group's greatest round-trip time and $GSIZE$ is the number of receivers in the multicast group. y is " the fraction of *(K * GRTT)* used to offset the backoff with respect to the sender's current transmission rate", and r is "the adjusted ratio of the local receiver's calculated rate to sender's current rate" [7]. Finally, the *RandomBackoff* function, given in Section 4.2.6, is the same one which is used by receivers in their NACK suppression mechanism in the data and repair transmission component.

The algorithm is implemented by the function `RTTbackoff` in the functional module `TIMEOUT-FUNCTIONS`:

```
*** Usage: RTTbackoff(seed, NormRobustFactor, GRTT,
***                   GSIZE, rcvRateInKbps, sndRateInKbps)
op RTTbackoff : Nat NzNat Nat NzNat Rat Rat -> NzNat .

eq RTTbackoff(N, NZN, N', NZN', R, R') =
  nat(1/4 * r(R, R') * maxBackoff(NZN, N')
      + (1 - 1/4) * randomBackoff(N, NZN, N', NZN')) .
```

y is set to 0.25, as recommended by the authors.

There are two definitions of r in the specification: One for the "slow start" phase of the congestion control component, when the sender hasn't received any reports about packet loss from the receiver set, and one for steady-state congestion control. Both functions result in values in the range 0-1. In my simplified model of NORM, I choose to use only the steady-state definition of r:

```
op r : Rat Rat -> Rat .
eq r(R, R') = (maximum(minimum((R / R'), 9/10), 1/2) - 1/2) / 2/5 .
```

The definitions of `minimum` and `maximum` can be found in Appendix B.

5.8 `delta` and `mte` for the `RTTsender` and `RTTreceiver` Classes

The following equations define the effect of time on the objects in a RTT configuration (`delta`) and the maximal possible time elapse in such a configuration (`mte`). The function `delta` increases the value of an object's clock and decreases the values of its timers with the elapsed time.

```
eq delta(< S : RTTsenderAlone | clock : T, CCtransTimer : TI >, T') =
         < S : RTTsenderAlone | clock : T + T',
                                CCtransTimer : TI monus T' > .

eq delta(< R : RTTreceiverAlone | clock : T, ACKtimer : TI,
                                  ACKholdoffTimer : TI',
                                  nacksToBeSent : ML >, T') =
         < R : RTTreceiverAlone | clock : T + T', ACKtimer : TI monus T',
                                  ACKholdoffTimer : TI' monus T',
                                  nacksToBeSent : delta(ML, T') > .

eq mte(< S : RTTsenderAlone | CCtransTimer : TI >) = TI .

eq mte(< R : RTTreceiverAlone | ACKtimer : TI, ACKholdoffTimer : TI',
                                nacksToBeSent : ML >) =
    if TI == INF and TI' == INF then INF
    else min(min(TI, TI'), leastDly(ML)) fi .
```

Chapter 6

Analysing the Round-Trip Time Component

In this chapter I will formally analyse the Real-Time Maude specification of the RTT component of the NORM protocol. The goal of the component is to keep a running estimate of the greatest round-trip time of the receiver group, and to notify the receivers of this estimate. My analysis indicates that, given relatively stable RTT values among the receivers, the protocol calculates a GRTT close to the largest RTT value of the group, and that the receivers get this estimate. However, the analysis reveals that the algorithm for calculating the GRTT estimate does not fully specify what should be done in certain cases.

For the analysis, I define two initial states, corresponding to two different network topologies. The first one is very simple: it has four nodes, and the GRTT estimate is easy to find because the receiver RTTs are the same. In the second initial state, which has six nodes, there are several different round-trip times, and estimating the GRTT is more complicated. Analysing the RTT component from two different initial states increases our confidence in the correctness of the protocol. It also demonstrates how easy it is to define new topologies in Real-Time Maude, in particular in comparison to testbeds. During a run of the protocol from either of the initial states, there will be no traffic in the links other than the messages used for round-trip time measurement. I therefore expect the RTT values for the receivers to be stable.

6.1 A Simple Initial State: `rtt1`

The first initial state I define is quite simple—it has a sender and two receivers, and a router which forwards messages on their behalf. The initial state `rtt1` is defined as follows in the module `NORM-RTT-1`, which imports the module `NORM-RTT`:

```
op rtt1 :  -> GlobalSystem .
eq rtt1 =
 {< sender : RTTsenderAlone | children : router, clock : 0,
               NormRobustFactor : 4, GRTT : 500, gsize : 2, sendRate : 50,
               sendRateInKbps : 256, CCtransTimer : INF,
               peakRTT : 0, CLRresponse : false, lowPeakRTTcounter : 0 >
    < router : Router | parent : sender, children : rec1 rec2, buffer : nil,
               bufferCap : 5, queuingDelay : 3 >
    < rec1 : RTTreceiverAlone | randomSeed : 799, RCV-ATTS >
    < rec2 : RTTreceiverAlone | randomSeed : 173, RCV-ATTS >
    < sender-router : Link | upNode : sender, downNode : router,
               propDelay : 14, LINK-ATTS >
    < router-rec1 : Link | upNode : router, downNode : rec1,
               propDelay : 19, LINK-ATTS >
    < router-rec2 : Link | upNode : router, downNode : rec2,
               propDelay : 19, LINK-ATTS >} .
```

RCV-ATTS denotes the receiver attributes

```
parent : router, clock : 0, NormRobustFactor : 4, GRTT : 0, gsize : 2,
ACKtimer : INF, ACKholdoffTimer : INF, timestamp : 0, receivedTimestamp : 0,
rcvRateInKbps : 256, sndRateInKbps : 0, CLR : false, nacksToBeSent : nil
```

and *LINK-ATTS* denotes the link attributes

```
upstream : nil, downstream : nil, bandwidth : 3
```

The names of the objects in the initial state—`sender`, `rec1`, `rec2`, and so on—are defined as constants of sort `Oid`. Figure 6.1 shows the topology of the initial state, and the delay values of the router and the (empty) links. The figure also shows the round-trip time values of the receivers. They both have the *same* RTT value, 76, to make it easier to make a first analysis of the sender's GRTT estimate. The RTT value of `rec1` and `rec2` is the time it takes for a control packet to travel from the sender to the receiver and back again, given that the links are empty. Neither of the receivers have the status of current limiting receiver (CLR), which means that both receivers (unless the `RTTbackoff` function returns 0) will suppress their feedback. The `NormRobustFactor` attribute of the sender and the receivers is set to 4, which is the value recommended in the NORM specification for any-source multicast.

6.2 Analysing *One* Possible Behaviour of the RTT Component from State `rtt1`

What sort of GRTT value can we expect from the initial state defined above? The sender's initial GRTT estimate is 500 ms, as specified by the NORM draft specification. When RTT values start coming in from the receivers, the sender calculates a new GRTT estimate according to the else-clause of the update function, which I repeat below:

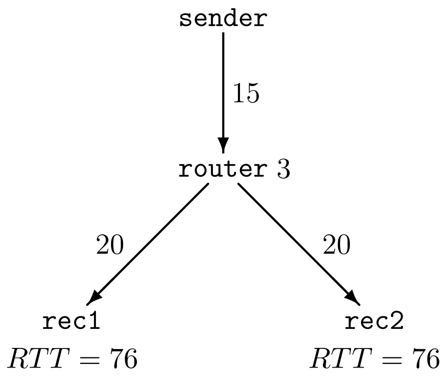

Figure 6.1: The topology corresponding to the initial state `rtt1`

```
if (peak > current_estimate)
    current_estimate = 0.25 * current_estimate + 0.75 * peak;
else
    current_estimate = 0.75 * current_estimate + 0.25 * peak;
```

As long as the peak RTT value recorded during a probing period (i.e., the interval between transmission of `CC` messages) is *less than or equal to* the current GRTT estimate, the function's else-clause will be used to calculate the new GRTT estimate value. Since the function then keeps a larger part of the old GRTT estimate than of the peak RTT, the new estimate value will always be greater than (or equal to) the peak RTT of the *next* probing interval when running the protocol from initial state `rtt1`. Consequently, since both receivers have the same RTT value, 76, I expect the initial GRTT estimate to be gradually reduced until it reaches a value slightly larger than the RTT value. I use Real-Time Maude's rewriting command to analyse *one* behaviour of the system from initial state `rtt1` up to time 15000:

```
Maude> (trew rtt1 in time <= 15000 .)

Result ClockedSystem :
 {< sender : RTTsenderAlone | CCtransTimer : 60, CLRresponse : false,
        GRTT : 78, NormRobustFactor : 4, children : router, clock : 14994,
        gsize : 2, lowPeakRTTcounter : 1, peakRTT : 76, sendRateInKbps : 256,
        sendRate : 50 >
  < sender-router : Link | bandwidth : 3, downNode : router, downstream : nil,
        propDelay : 14, upNode : sender, upstream : nil >
  < router : Router | bufferCap : 5, buffer : nil, children :(rec1 rec2),
        parent : sender, queuingDelay : 3 >
  < router-rec1 : Link | bandwidth : 3, downNode : rec1,
        downstream : dly(CC(14976,78,256),20), propDelay : 19,
        upNode : router, upstream : nil >
  < router-rec2 : Link | bandwidth : 3, downNode : rec2,
        downstream : dly(CC(14976,78,256),20), propDelay : 19,
        upNode : router, upstream : nil >
  < rec1 : RTTreceiverAlone | ACKholdoffTimer : INF, ACKtimer : 72,
        CLR : false, GRTT : 78, NormRobustFactor : 4, clock : 14994,
        gsize : 2, nacksToBeSent : nil, parent : router, randomSeed : 5171,
        rcvRateInKbps : 256, receivedTimestamp : 14780, sndRateInKbps : 256,
        timestamp : 14742 >
  < rec2 : RTTreceiverAlone | ACKholdoffTimer : INF, ACKtimer : 11,
        CLR : false, GRTT : 78, NormRobustFactor : 4, clock : 14994,
        gsize : 2, nacksToBeSent : nil, parent : router, randomSeed : 1970,
        rcvRateInKbps : 256, receivedTimestamp : 14780, sndRateInKbps : 256,
        timestamp : 14742 >} in time 14994
```

The sender's GRTT estimate is 78 ms, which is the kind of value I expected. The sender's peakRTT attribute shows the largest RTT value it has received from the receivers during the current probing interval—76. Furthermore, both receivers have received the GRTT estimate. Will the GRTT estimate be further reduced? Another rewrite of the initial state, this time with a time bound of 25000, results in a state where the GRTT is 78, as before[1]:

```
Maude> (trew rtt1 in time <= 25000 .)

Result ClockedSystem :
 {< sender : RTTsenderAlone | GRTT : 78, peakRTT : 77,
              lowPeakRTTcounter : 0, CLRresponse : false, [...] >
  < rec1 : RTTreceiverAlone | GRTT : 78, [...] >
  < rec2 : RTTreceiverAlone | GRTT : 78, [...] >
  [...]} in time 24998
```

However, the peak RTT value as recorded by the sender is now 77, even though I expected this value to be stable at 76. It would be interesting to analyse closer the behaviour simulated in the above rewrite command. We can use Real-Time Maude's *tracing* facilities to output all the rewrite steps in the above behaviour. I put the following tracing commands at the start of the file containing the module NORM-RTT-1, where the initial state rtt1 is defined[2],

```
  set trace on .
  trace exclude REAL-TIME-MAUDE .
  set trace substitution off .
  set trace eq off .
```

and add the rewrite command with time bound 25000 at the end of the file. When executing the file containing the rewrite and tracing commands, I redirect the resulting trace to a new

[1] To make the result from the rewrite and search commands more readable, I replace some of the ouput with [...].

[2] These commands can also be given at the Maude prompt.

file. The trace is fairly large, and it is more convenient to ouput it to a file to analyse the behaviour. The trace contains the following state, encountered at time 21581, in which two receiver feedback messages have entered the same link:

```
 {< sender : RTTsenderAlone | GRTT : 78, peakRTT : 76, [...] >
  < sender-router : Link | propDelay : 14, upNode : sender,
         upstream : (dly(ACK(21520, 256, false), 15)
                     ++ dly(ACK(21520, 256, false), 16)), [...] >
  [...]}  in time 21581
```

`rec1` and `rec2` have each sent an `ACK` message to the sender at the same time. The first message to enter the link from the router up to the sender is given a delay value of 15 ms, which is the delay of a control message in this link when it is otherwise empty. The *second* `ACK` message enters a link which is *not* empty, and is given a larger delay value—16 ms. Consequently, when the second feedback message reaches the sender, it has an RTT value of 77, and not 76 like the message which entered an empty link. This behaviour is completely according to what can be expected when a link contains more than one message, so that our analysis of the potentially spurious behaviour did not reveal any error in the specification.

6.3 Model Checking the RTT Component from State `rtt1`

In this section I use model checking techniques to analyse not just *one* behaviour from the initial state, but *all* behaviours—up to a certain time limit to ensure termination of the model checking—from the initial state.

I start this section by continuing the analysis of the above setting, in which a link contains more than one message, and where the transmission delay of a message which enters a non-empty link is larger than the minimal transmission delay. This is not an uncommon situation. However, it is far more usual that the link between the router and the sender only contains one message at a time[3]. Thus, most of the time, the RTT values for the receivers will be 76 ms, and it seems reasonable to expect that the sender's `peakRTT` attribute will return to the value 76 after a while. Examining the tracing result from the previous section further shows that the first state where the sender's `peakRTT` attribute has the value 77 is encountered at time 21597. A search from this state—which I call `rtt1b`—shows that the peak RTT value does *not* return to 76:

```
Maude> (tsearch rtt1b =>*
           {< sender : RTTsenderAlone | peakRTT : 76, ATTS:AttributeSet >
            REST:Configuration}
             in time <= 30000 .)
```

[3] A simple search from the initial state, using the `tsearch` command with a time bound of 30000 ms, results in 63 possible states where there are two messages traveling upstream through the `sender-router` link simultaneously. On the other hand, searching for states where there is only one message in the link, yields 1373 solutions.

```
No solution
```

This behaviour can be traced back to the rules for updating the GRTT estimate, which are specified in Section 3.7.1 in the building block document [6]. The sender always records the highest RTT value it receives during a probing interval. If it receives a RTT which is greater than both the current GRTT estimate *and* the current peak RTT, the GRTT is updated immediately. Otherwise, it tracks low RTT values across three consecutive intervals before the GRTT is updated. At the *end* of a probing interval, the peak RTT tracking value is either

1. set to zero if it is greater than or equal to the current GRTT estimate (i.e., it has already been used for updating the GRTT), or
2. kept the same if it is less than the current GRTT estimate and this is *not* the third consecutive probing interval with a low RTT tracking value.

These rules do not specify what to do if the peak RTT value is less than the current GRTT estimate and this *is* the third consecutive probing interval. For lack of a rule for this case, the rewrite rule for GRTT updating at the end of a probing interval, `[endOfRTTcollectionPeriod]`, simply keeps the low RTT value. In the example above, where there are only low RTT values in the system, the sender's `peakRTT` attribute will not change once it reaches the value 77—this will remain the peak RTT value. This can be confirmed by model checking the RTT specification w.r.t. the initial state `rtt1` to see if the `peakRTT` attribute will always be 77 once it reaches that value. The atomic proposition `peakRTTis77` is defined to be true in every state where the sender's `peakRTT` attribute is 77 :

```
(tomod MODEL-CHECK-NORM-RTT-1 is
  including TIMED-MODEL-CHECKER .
  protecting NORM-RTT-1 .

  op peakRTTis77 : -> Prop [ctor] .
  eq {< sender : RTTsenderAlone | peakRTT : 77 >  REST:Configuration}
           |= peakRTTis77 = true .
endtom)
```

The time-bounded[4] temporal logic model checking command below can show that *if* the sender's RTT tracking value reaches 77, it will not change afterwards[5]:

```
Maude> (mc rtt1 |=t peakRTTis77 => [] peakRTTis77 in time < 100000 .)

Result Bool : true
```

[4] Since the reachable state space from `rtt1` if infinite, we must use time-bounded model checking to ensure termination.

[5] In the Maude model checker, `A => B` is defined as `[] (A -> B)` [14].

Does this affect the sender's GRTT estimate w.r.t. the initial state `rtt1`? The GRTT value 78 ms appeared to be the lowest GRTT that the sender calculates. The model checker can be used to find out if there is a state in every behaviour where the GRTT estimate reaches 78, and if this estimate is stable throughout the rest of the behaviour. At the same time, we can check that the receivers have received the estimate. The atomic proposistion `GRTTis78` holds in every state where the `GRTT` attribute of both the sender and the receivers has the value 78:

```
  op GRTTis78 : -> Prop [ctor] .
  eq {< sender : RTTsenderAlone | GRTT : 78 >
      < rec1 : RTTreceiverAlone | GRTT : 78 >
      < rec2 : RTTreceiverAlone | GRTT : 78 >  REST:Configuration}
            |= GRTTis78 = true .
```

The model checking command confirms that there is a state in *every* behaviour where the sender and the receivers have a GRTT value of 78, and that the GRTT remains the same throughout the rest of the behaviour:

```
Maude> (mc rtt1 |=t (<> GRTTis78 /\ (GRTTis78 => [] GRTTis78))
          in time < 100000 .)

Result Bool : true
```

6.4 A Larger Initial State: `rtt2`

Although `rtt1` was a very simple initial state, analysing the RTT specification from it produced some interesting results. However, it is also important to analyse the RTT component from a more complex initial state where there are several different round-trip time values in the receiver group. The initial state `rtt2`, defined in the module `NORM-RTT-2`, has one sender, three receivers, and two routers, which are connected by five link objects:

```
  op rtt2 : -> GlobalSystem .
  eq rtt2 =
   {< sender : RTTsenderAlone | children : router1, clock : 0,
               NormRobustFactor : 4, GRTT : 500, gsize : 3, sendRate : 40,
               sendRateInKbps : 256, CCtransTimer : INF, peakRTT : 0,
               CLRresponse : false, lowPeakRTTcounter : 0 >
    < router1 : Router | parent : sender, children : rec1 router2, RTR-ATTS >
    < router2 : Router | parent : router1, children : rec2 rec3, RTR-ATTS >
    < rec1 : RTTreceiverAlone | parent : router1,  randomSeed : 779,
               CLR : false, RCV-ATTS >
    < rec2 : RTTreceiverAlone | parent : router2, randomSeed : 13,
               CLR : true, RCV-ATTS >
    < rec3 : RTTreceiverAlone | parent : router2, randomSeed : 9,
                CLR : false, RCV-ATTS >
    < sender-router1 : Link | upNode : sender, downNode : router1,
                propDelay : 5, bandwidth : 10, LINK-ATTS >
    < router1-rec1 : Link | upNode : router1, downNode : rec1,
                propDelay : 20, bandwidth : 1, LINK-ATTS >
    < router1-router2 : Link | upNode : router1, downNode : router2,
                propDelay : 21, bandwidth : 1, LINK-ATTS >
    < router2-rec2 : Link | upNode : router2, downNode : rec2,
                propDelay : 10, bandwidth : 3, LINK-ATTS >
    < router2-rec3 : Link | upNode : router2, downNode : rec3,
                propDelay : 7, bandwidth : 5, LINK-ATTS >} .
```

RTR-ATTS denotes the attributes that the router objects have in common:

```
  buffer : nil, bufferCap : 5, queuingDelay : 3
```

RCV-ATTS denotes the common receiver attributes:

```
  clock : 0, NormRobustFactor : 4, GRTT : 0, gsize : 3,  ACKtimer : INF,
  ACKholdoffTimer : INF, timestamp : 0, receivedTimestamp : 0,
  rcvRateInKbps : 256, sndRateInKbps : 0, nacksToBeSent : nil
```

Finally, *LINK-ATTS* denotes the common attributes of the link objects:

```
  upstream : nil, downstream : nil
```

Figure 6.2 shows the topology of the state, the delay values of the routers and links, and the round-trip times of the receivers. As previously, the RTT values are based on control messages traveling through otherwise empty links. rec2, with an RTT value of 90 ms, is the current limiting receiver, and its CLR attribute is set to true.

6.5 Analysing *One* Possible Behaviour of the RTT Component from State rtt2

As with the previous initial state, I start by rewriting rtt2 to see what the GRTT estimate may be after approximately 15000 ms:

```
Maude> (trew rtt2 in time <= 15000 .)

Result ClockedSystem :
 {< sender : RTTsenderAlone | GRTT : 92, peakRTT : 90,
```

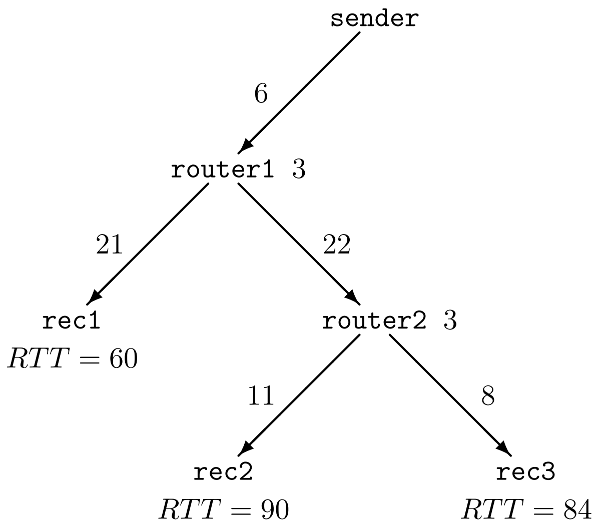

Figure 6.2: The topology corresponding to the initial state `rtt2`

```
              lowPeakRTTcounter : 0, CLRresponse : true, [...] >
  < rec1 : RTTreceiverAlone | GRTT : 92, [...] >
  < rec2 : RTTreceiverAlone | GRTT : 92, [...] >
  < rec3 : RTTreceiverAlone | GRTT : 92, [...] >
  [...]} in time 14999
```

The sender's `GRTT` attribute has the value 92 at time 14999, which is a promising result. Because the largest RTT value in the group is 90 and the topology is stable, we can expect the GRTT estimate to be somewhat larger than 90. Another rewrite sequence returns a state at time 24995 where the GRTT is also 92:

```
Maude> (trew rtt2 in time <= 25000 .)

Result ClockedSystem :
 {< sender : RTTsenderAlone | GRTT : 92, peakRTT : 90,
             lowPeakRTTcounter : 0, CLRresponse : true, [...] >
  < rec1 : RTTreceiverAlone | GRTT : 92, [...] >
  < rec2 : RTTreceiverAlone | GRTT : 92, [...] >
  < rec3 : RTTreceiverAlone | GRTT : 92, [...] >
  [...]} in time 24995
```

Unlike `rtt1`, rewriting `rtt2` did not produce a behaviour where the peak RTT changes after a while. The sender has recorded the current limiting receiver's RTT value of 90 ms in its `peakRTT` attribute, and no larger value has been registered at time 24995. However, this is just *one* of the possible behaviours of the system from `rtt2`. There *may* be other behaviours where two or more feedback messages enter the same link on their way to the sender, causing the peak RTT to become larger than 90 ms. I will return to this question in the course of the next section.

6.6 Model Checking the RTT Component from State `rtt2`

Before investigating possible changes in the peak RTT value, I would like to know how long it takes to reach the GRTT value found in the previous section. The major difference between the two initial states I have defined in this chapter is that while `rec2` is denoted the current limiting receiver in `rtt2`, there is no CLR in `rtt1`. Does this have a significant effect on the time it takes to initialise the GRTT estimate in the RTT component? Searching for the first occurence of a state where the sender's GRTT estimate is 92 gives the following result:

```
Maude> (find earliest rtt2 =>*
           {< sender : RTTsenderAlone | GRTT : 92, ATTS:AttributeSet >
               REST:Configuration} .)

Result:
 {< sender : RTTsenderAlone | GRTT : 92, [...] >
  < rec1 : RTTreceiverAlone | GRTT : 93, [...] >
  < rec2 : RTTreceiverAlone | GRTT : 93, [...] >
  < rec3 : RTTreceiverAlone | GRTT : 93, [...] >
  [...]} in time 5540
```

A similar search from the initial state `rtt1` shows that it takes 21597 ms before the sender reaches its lowest GRTT estimate. Thus, the system w.r.t. initial state `rtt2` reached what seems to be its lowest GRTT estimate in one fourth of the time it took to reach the corresponding estimate from `rtt1`. By searching from `rtt2` for the first occurence of a state where the sender's `peakRTT` attribute has a nonzero value, we can find out how long it takes before the sender receives any RTT feedback:

```
Maude> (find earliest rtt2 =>*
          {< sender : RTTsenderAlone | peakRTT : NZN:NzNat, ATTS:AttributeSet >
              REST:Configuration} .)

Result:
 {< sender : RTTsenderAlone | GRTT : 500, peakRTT : 90, lowPeakRTTcounter : 0,
            CLRresponse : true, [...] >
  [...]} in time 90
```

The current limiting receiver, `rec2`, replied immediately to the sender's `CC` message. The sender received a reply to its message after 90 ms—which is the RTT value for `rec2`. Both receivers in `rtt1` use a backoff timeout before sending a `ACK`, and a similar search from this initial state shows that is takes 1520 ms before the first RTT feedback message reaches the sender. Although the state `rtt1` is artificial because there *should* be a receiver with CLR status, the difference in response time between the two states can serve to demonstrate the purpose of the current limiting receiver.

Let us return now to the peak RTT value—will there ever be a peak RTT higher than 90 ms? The atomic proposition `peakRTTis90` defined below holds in every state where the sender's `peakRTT` attribute has the value 90:

```
(tomod MODEL-CHECK-NORM-RTT-2 is
  including TIMED-MODEL-CHECKER .
  protecting NORM-RTT-2 .

  op peakRTTis90 : -> Prop [ctor] .
  eq {< sender : RTTsenderAlone | peakRTT : 90 >  REST:Configuration}
          |= peakRTTis90 = true .
endtom)
```

Model checking the RTT specification w.r.t. the initial state `rtt2` shows that if the sender has recorded the RTT value 90, it will not receive any higher values. The situation that arose in the system with the first initial state does not appear here.

```
Maude> (mc rtt2 |=t peakRTTis90 => [] peakRTTis90 in time < 70000 .)

Result Bool : true
```

Finally, I want to check if the GRTT estimate will eventually appear in *every* possible behaviour, and that once it appears, it will remain stable. The proposition `GRTTis92` holds in every state where the `GRTT` attribute of the sender and the receivers has the value 92:

```
  op GRTTis92 : -> Prop [ctor] .
  eq {< sender : RTTsenderAlone | GRTT : 92 >
      < rec1 : RTTreceiverAlone | GRTT : 92 >
      < rec2 : RTTreceiverAlone | GRTT : 92 >
      < rec3 : RTTreceiverAlone | GRTT : 92 >  REST:Configuration}
            |= GRTTis92 = true .
```

The model checking command confirms that this until-stable property holds for the RTT component w.r.t. the initial state `rtt2`:

```
Maude> (mc rtt2 |=t (<> GRTTis92 /\ (GRTTis92 => [] GRTTis92))
              in time < 70000 .)

Result Bool : true
```

6.7 Concluding Remarks

Analysing the RTT component from the initial state `rtt1` showed that the GRTT updating algorithm does not fully specify what the sender should do with its peak RTT value once it has been used for updating the GRTT estimate. A closer look at the algorithm also reveals that case one, where the peak RTT is set to zero at the end of a probing interval if it is greater than or equal to the current GRTT estimate, is problematic: The algorithm does not specify what the sender should do at the end of the *next* interval if it has not received any new RTT values so that its peak RTT is still zero. Should it update the GRTT with zero? Should it *not* update the GRTT?

Although neither of these problems had an effect on the correctness of the RTT protocol w.r.t. the initial states `rtt1` and `rtt2`, it is possible they will influence the GRTT estimate in dynamic topologies. The GRTT estimate is in part used in the receiver feedback suppression mechanisms of the NORM protocol, and the purpose of these suppression mechanisms is to allow the protocol to scale to large groups of receivers by limiting the feedback volume. Thus, the cases of underspecification in the RTT component might possibly have some influence on the scalability of NORM, since the suppression mechanisms will not function optimally.

In the latest version of the NORM building block document [8], the algorithm has been substantially changed. Among other things, the sender no longer tracks low RTT values across three intervals, but updates the GRTT with the peak RTT value it received during each interval. The peak is always set to zero at the end of an interval, and if there is no feedback during an interval, the GRTT remains unchanged.

To summarise, the analysis of the RTT component showed that

- the sender calculated a greatest round-trip time estimate close to the recorded peak RTT value of the receiver group,
- the current GRTT estimate was multicast to the receivers, and
- it is not always clear how the GRTT estimate and the peak RTT value should be updated.

In addition, the analysis has demonstrated how the use of formal modelling and analysis techniques can bring to light which parts of a specification that are not fully specified.

Chapter 7

Specifying the Data and Repair Transmission Component

We now move on to the modelling of the data and repair transmission component. This part of the specification is much more complex than the RTT component, and there will be more decisions to make. I will start by indentifying some problematic areas of the informal NORM specification by describing ambiguities, inconsistencies, and cases of underspecification in the text. Then I model the component on the basis of the decisions I make concerning these problems. The module `APPLICATIONS` defines an application level for the component. In the module `NORM-DATA-TRANSMISSION` I declare the sender and receiver classes, and specify the protocol procedures. The rewrite rules use a number of functions, whose definitions will not be given in this chapter, but which can be found in Appendix B.

7.1 Identifying Ambiguities

The process of formalising brings out ambiguities and implicit assumptions in a specification. This section indentifies some parts of the component which are ambiguous or which appear not to be fully specified. Although my model will be entirely based on the two documents I have used so far, I will in some cases look at the latest versions of the specification to see what has been done.

7.1.1 Receiver NACK Cycle Initiation

The NORM specification's description of how a receiver initiates a NACK cycle is straightforward: "The NACKing procedure SHALL be initiated *only* at NormObject boundaries, FEC coding block boundaries, or upon receipt of a NORM_CMD(FLUSH) [...][1] message" [7]. The building block document comments that limiting NACK process initiation to specific events helps reduce the number of repair request messages sent—the receivers aggregate their requests

[1] I have omitted an option which is used when the receivers can only unicast their feedback.

into a smaller number of NACK messages, and there is a loose degree of synchronisation among receivers with repair needs that helps make the feedback suppression mechanism more effective [6]. The specification also says that after generating a NACK message, the receiver will wait an appropriate amount of time before repeating the NACK if it did not receive the necessary repair segments. Given the specification's description of how receivers initiate NACK cycles, it seems natural to interpret this as meaning that the receiver will start a new NACK backoff period at the next NACK cycle initiation event, provided its holdoff timeout is over.

However, Section 4.2.3 of the NORM specification mentions another possibility: when no messages are received from the sender, a receiver can *self-initiate* the repair request procedure after an inactivity timeout "related to $2 * GRTT * NORM\_ROBUST\_FACTOR$" [7]. The specification says that this option will be discussed more later, but there is no further reference to it in either the specification or the building block document. As far as I can see, the latest version of the specification [5] does not provide any more details.

The procedure of NACK initiation at specific events is carefully described, and I can easily model it in a Real-Time Maude specification. As a consequence of my decisions in Chapter 4 to omit forward error correction, I restrict the NACK events to detection of object boundaries and reception of a FLUSH message. However, I know to little about the procedure of self-initiating NACK cycles to be able to safely include it in the formal model.

7.1.2 Sender NACK Accumulation Timeout

When a sender gets a NACK message from a receiver, it starts its repair procedure with a repair request aggregation timeout. The NORM specification sets the duration of this aggregation period to $K * GRTT$, where K is the constant factor used in timeout functions throughout the protocol (the NORM robust factor) [7]. When the timeout expires, the sender begins transmitting repair content.

In its description of the sender repair process, the building block document sets the timeout to at least $(K + 1) * GRTT$. The duration of the sender NACK aggregation period should be sufficient to allow the receivers to use their feedback suppression mechanisms, and any resulting NACK messages to propagate through the network. Thus the timeout value needs to be larger than the receiver feedback suppression algorithm's maximum allowed backoff timeout of $K * GRTT$ (described in Section 4.2.6).

In the latest version of the NORM protocol specification [5], the sender timeout value is set to $(K + 1) * GRTT$. Since this seems to be the original intention, this is what I will use in my model.

7.1.3 Sender FLUSH Process

When a sender reaches the end of newly available transmit content, both new data and repair content, it transmits a series of NORM_CMD(FLUSH) messages to invite the receivers to NACK for missing data content up to and including an announced transmission point. The command is repeated once per $2 * GRTT$, and the number of repeats is equal to the NORM robust factor value. If flushing is interrupted, the sender will reinitiate the process once it has performed the requested repairs [7].

The intention seems to be that the sender gives the receivers a chance to ask for repair of missing segments before it clears the data content to make room for new data objects enqueued for transmission by the application program. The repetition of the FLUSH and the $2 * GRTT$ interval between each message will give the receivers an opportunity to ask for repair even under poor network conditions.

From a logical point of view, given that the meaning of a FLUSH message is that a receiver *should* begin its NACK cycle if it needs repair, one would expect that the sender waits a certain (amount of) time before clearing the data content, once the last FLUSH has been sent. After all, there could be a receiver in the group that did not get any of the other FLUSH messages, and that consequently did not have the chance to initiate its NACK process before it received the very last FLUSH.

Neither the NORM specification nor the building block document mentions such a "safety/final chance period". It could be that solutions to such protocol functions are self-evident to a network expert, but a non-expert is forced to base her understanding of the protocol entirely on what is written in the specification. As a consequence, a sender in my model of the NORM protocol will simply clear the buffered data objects immediately once it has sent its last FLUSH.

Another part of the FLUSH process which is not entirely clear is the way in which the sender informs the receivers that the session is about to close. The NORM specification says that "When the NORM_FLUSH_FLAG_EOT flag is set, this indicates the sender is preparing to terminate transmission and will no longer provide response to repair requests" [7]. What exactly does this mean? Has the sender already sent a series of FLUSH messages (with a false eot flag) and performed the necessary repairs, so that these last end-of-transmission FLUSHes are sent, not to invite the receivers to request further repair, but just to inform them that transmission is over? Or does the quoted sentence mean that the last series of FLUSH messages, with the eot flag set, is followed by the usual repair procedures? I opt for this second interpretation in the model.

7.2 The DTsender and DTreceiver Classes

The class `DTsender` is declared as follows:

```
class DTsender |
  dataBuffer : Msg,
  flushBuffer : Msg,
  repairTransmission : DataUnitIdList,
  currentTransPos : DataUnitId,
  lastNewDataId : DataUnitId,
  lastRepairDataId : DataUnitId,
  dataTransTimer : TimeInf,          *** determinded by sendRate
```

```
    accNACKcontent : DataUnitIdList,
    invalidRepairRequests : DataUnitIdList,
    NACKaccumTimer : TimeInf,          *** (K+1)*GRTT
    repairCycleHoldoffTimer : TimeInf, *** 1*GRTT
    FLUSHtimer : TimeInf,              *** 2*GRTT
    FLUSHcounter : Nat,
    SQUELCHholdoffTimer : TimeInf .    *** 2*GRTT
```

The attributes `dataBuffer` and `flushBuffer` contain, respectively, new data content currently being transmitted, and content which has been transmitted and is kept for a certain time in case there are any repair requests. `currentTransPos` records the current transmission position, regardless of whether it concerns new or repair data content. `lastNewDataId` and `lastRepairDataId` record the sender's ordinary and repair transmission positions. The pace of transmission is determined by the `dataTransTimer`. During the NACK accumulation period, the aggregated repair requests are kept in `accNACKcontent`, and when the `NACKaccumTimer` expires, the content is moved to `repairTransmission` and the repair transmission begins. After a NACK accumulation period, the `repairCycleHoldoffTimer` is set. `FLUSHtimer` and `FLUSHcounter` are used in the sending of `FLUSH` messages, and `SQUELCHholdoffTimer` is set after a `SQUELCH` has been sent.

Similarly, I declare the class `DTreceiver`:

```
  class DTreceiver |
    receiveBuffer : MsgList,
    repairNeeds : DataUnitIdList,
    sendersCurrTransPos : DataUnitId,
    nextExpectedDUI : DataUnitId,
    repairRequests : DataUnitIdList,
    extRepRequests : DataUnitIdList,
    NACKbackoffTimer : TimeInf,
    NACKcycleHoldoffTimer : TimeInf .
```

The DTreceiver puts the data content it receives in the `receiveBuffer` and records missing segments in `repairNeeds`. `nextExpectedDUI` identifies the next expected new data segment, while `sendersCurrTransPos` keeps track of the sender's transmission position. During a NACK cycle, `repairRequests` contains the identifiers of the missing segments the receiver would like to request, and requests heard from other receivers are kept in `extRepRequests`. The duration of the NACK suppression timeout is controlled by `NACKbackoffTimer`, and while the `NACKcycleHoldoffTimer` is running, the receiver cannot initiate a new NACK cycle.

In addition to the attributes declared above, both classes inherit the following attributes from their superclasses: `clock`, `NormRobustFactor`, `GRTT` and `gsize`. The `DTsender` class also inherits the attributes `children` and `sendRate`, while the `DTreceiver` inherits `parent`, `randomSeed` and `CLR`:

```
  subclass DTsender < Sender .
  subclass DTreceiver < Receiver .
```

Finally, I declare the subclasses DTsenderAlone and DTreceiverAlone for stand-alone execution of the component:

```
class DTsenderAlone .      subclass DTsenderAlone < DTsender .
class DTreceiverAlone .    subclass DTreceiverAlone < DTreceiver .
```

7.3 Variable Declarations for the Component

As in Chapters 4 and 5, I list the variable declarations for the modules NORM-DATA-TRANSMISSION and APPLICATIONS:

```
vars A A' S R O : Oid .
var OS : OidSet .
vars M M' : Msg .
var DP DP' DP'' : DataPacket .
vars ML ML' ML'' : MsgList .
vars DUI DUI' DUI'' : DataUnitId .
vars DUIL DUIL' DUIL'' : DataUnitIdList .
vars T T' : Time .
vars TI TI' TI'' TI''' : TimeInf .
vars N N' N'' N''' : Nat .
vars NZN NZN' NZN'' NZN''' NZN'''' NZN''''' NZN'''''' : NzNat .
vars B B' : Bool .
```

7.4 The Application Level

A NORM protocol node transmits and receives data content on behalf of an application on the local machine. The module APPLICATIONS defines the classes of sender and receiver applications. These are very simple: The sender application enqueues data content for transmission during a session, and the receiver application collects received data from the receiver node. Since there will usually be more than one receiver application in a NORM session, each receiver object identifies its NORM protocol receiver in the receiver attribute.

```
class SenderApplication | dataBuffer : MsgList .

class ReceiverApplication | receiver : Oid, dataBuffer : DataUnitIdList .
```

I add equations for delta and mte:

```
eq delta(< A : SenderApplication | >, T) = < A : SenderApplication | > .
eq mte(< A : SenderApplication | >) = INF .
eq delta(< A : ReceiverApplication | >, T) = < A : ReceiverApplication | > .
eq mte(< A : ReceiverApplication | >) = INF .
```

The interaction between the protocol and application layer in our model is very simple, and strictly speaking it is not really necessary to include it at all, as it does not influence the logic of the communcation between the protocol nodes. Still, having an application layer in the model is convenient, because it makes the specification more intuitive and easier to execute.

7.5 Data Content and Messages

Section 3.2.2 described how the sender transmits messages, or objects, on behalf of its application, and how these messages are identified by an object number. In my model, such an object is implemented as the message `OBJECT`. Each `OBJECT` has an object identifier, and contains information about the number of segments in the object and how many of these have been transmitted (it is not necessary to identify the sender of the object, because in this model there will only be one application which sends data):

```
*** Usage: OBJECT(objectId, noOfSegsInObject, noOfSegsTransmitted)
msg OBJECT : NzNat NzNat Nat -> Msg .
```

In an initial state for the protocol, the sender application's buffer contains one or more blocks of data content objects. These are transferred one by one to the application's protocol sender. Each object block contains a list of `OBJECT` messages. The constant `noObjectBlock` is used to mark that the protocol sender's data buffer or flush buffer is empty.

```
*** Usage: objectBlock(listOfOBJECTs)
msg objectBlock : MsgList -> Msg .
msg noObjectBlock : -> Msg .
```

The module `MESSAGES` from Section 4.2.4 defines the sender and receiver messages. The message `DATA` is used by the sender to transmit the segments of an object. In addition to the object and segment identifiers, the sender includes the total number of segments for the object. I am not sure whether this is correct with respect to the NORM specification. It may for instance be more correct to model a data content message with flags that inform the receiver that this is the first or last segment, or that more segments will arrive. Due to my choice about including the number of segments, the receiver may end up having more knowledge than is intended in the specification. To compensate for this, the receiver rules will not take advantage of this extra knowledge. The receiver will act as if it knows that a segment is the first, the last, or neither. `DATA`, like every sender message, also informs the receiver of the current GRTT estimate. If the repair flag is set, the message carries a repair segment.

```
*** Usage: DATA(dataUnitId, noOfSegmentsInObject, grtt, repairFlag)
msg DATA : DataUnitId NzNat Time Bool -> DataPacket .
```

The FLUSH message identifies the sender's current transmission position (receivers will check their repair needs up to and including this position), and has a flag that shows whether this is the end of transmission or not.

```
*** Usage: FLUSH(dataUnitId, grtt, eotFlag)
msg FLUSH : DataUnitId Time Bool -> ControlPacket .
```

The NORM specification says that when the sender receives a repair request for an object that is outdated, aborted, or otherwise, it sends a message to inform the receivers of the earliest transmission position for which it can provide repair, and which objects, if any, within the repair "window" that are invalid. In my model, it will not be possible to declare an object invalid when it is within the range of possible repairs, so my SQUELCH message is a simplification of the one defined in the informal specification—it simply states the earliest valid repair position.

```
*** Usage: SQUELCH(earliestDUIvalidForRepair, grtt)
msg SQUELCH : DataUnitId Time -> ControlPacket .
```

In specifying the RTT component, I used a constant NACK to simulate RTT feedback in conjunction with repair requests. In this chapter I use the NACK message from the module MESSAGES. It contains a list of repair requests, the adjusted sender timestamp (which will be set to zero in the rules for the stand-alone component), and a flag that shows whether this message came from the current limiting receiver:

```
*** Usage: NACK(dataUnitIdList, adjustedTimestamp, CLRflag)
msg NACK : DataUnitIdList Time Bool -> ControlPacket .
```

7.6 The DTsender rules

In this section, I model the sender actions as rewrite rules. I start with rules for transmission of data content, and flushing of transmitted content. I then model the NACK accumulation period and the resulting repair transmission. Finally, I specify rules for the repair cycle holdoff timeout, and for generation of SQUELCH messages.

7.6.1 Transmission of New Data Content

At the start of a NORM session, the sender application has a list of object blocks in its buffer. Whenever the protocol sender is ready to transmit new data content, i.e., its dataBuffer attribute is empty, the application can give it a new block of objects:

```
rl [nextObjectBlock] :
  < A : SenderApplication | dataBuffer : M ++ ML >
  < S : DTsender | dataBuffer : noObjectBlock >
  =>
  < A : SenderApplication | dataBuffer : ML >
  < S : DTsender | dataBuffer : M > .
```

I could of course instead let the `DTsender` object store all the data blocks that the application wants to send. Such a (actually slightly less general) model would of course not in any way affect the actual protocol or its analysis.

In the next rule the sender sends the first segment of the next object to be transmitted. At the same time it sets or resets (depending on whether this is the very first segment sent or it is in the middle of transmission) the `dataTransTimer`. The time interval between each `DATA` message is determined by the value of `sendRate`, which is set in the inital state.

```
crl [transmitFirstSegOfObject] :
  < S : DTsenderAlone |
    children : OS, GRTT : T,
    dataBuffer : objectBlock(ML ++ OBJECT(NZN, NZN', 0) ++ ML'),
    lastNewDataId : N :: N', sendRate : T', dataTransTimer : TI,
    repairTransmission : nil >
  =>
  < S : DTsenderAlone |
    currentTransPos : s(N) :: 1,
    dataBuffer : objectBlock(ML ++ OBJECT(NZN, NZN', 1) ++ ML'),
    lastNewDataId : s(N) :: 1, dataTransTimer : T' >
  (multisend DATA(s(N) :: 1, NZN', T, false) from S to OS)
  if s(N) == NZN /\ allSegsTransmitted(ML) /\ (TI == INF or TI == 0) .
```

Once the first segment has been sent, the following rule is used to transmit the rest of the object. When the last segment of the last object in the object block has been sent, the data transmission timer is turned off, regardless of whether the application has more data content enqueued. There may be a delay between the time the last segment was sent and the time a new object block comes from the application, and time will not advance if the timer value is 0.

```
crl [transmitRestOfObject] :
  < S : DTsender |
    children : OS, GRTT : T,
    dataBuffer : objectBlock(ML' ++ OBJECT(NZN, NZN', NZN'') ++ ML''),
    lastNewDataId : NZN :: NZN'', sendRate : T', dataTransTimer : 0,
    repairTransmission : nil >
  =>
  < S : DTsender |
    currentTransPos : NZN :: s(NZN''),
    dataBuffer :
      objectBlock(ML' ++ OBJECT(NZN, NZN', s(NZN'')) ++ ML''),
    lastNewDataId : NZN :: s(NZN''),
    dataTransTimer :
      (if ML'' == nil and s(NZN'') == NZN' then INF else T' fi) >
  (multisend DATA(NZN :: s(NZN''), NZN', T, false) from S to OS)
  if NZN'' < NZN' .
```

7.6.2 The Flush Process

When the sender reaches the end of enqueued new segments and repair segments, it starts the flush process in order to invite the receivers to request repair for missing data content. If there is no more data content to be sent, the end-of-transmission flag in the FLUSH message is set.

```
  crl [initiateFlushProcess] :
  {< A : SenderApplication | dataBuffer : ML >
    < S : DTsender |
      children : OS, GRTT : T,
      dataBuffer : objectBlock(ML'), flushBuffer : noObjectBlock,
      FLUSHcounter : 0, FLUSHtimer : INF,
      NACKaccumTimer : INF, repairTransmission : nil >
    OC:ObjectConfiguration}
    =>
  {< A : SenderApplication | >
    < S : DTsender |
      dataBuffer : noObjectBlock, flushBuffer : objectBlock(ML'),
      FLUSHcounter : 1, FLUSHtimer : 2 * T >
    (multisend FLUSH(lastDataUnitId(objectBlock(ML')), T,
                     (if ML == nil then true else false fi))
     from S to OS)
    OC:ObjectConfiguration}
    if allSegsTransmitted(ML') .
```

Once the flush process is started, the following rule takes care of sending the remaining FLUSH messages. The number of messages sent is determined by the NORM robust factor. When the final FLUSH is sent, the content of the flushBuffer is cleared. This rule is also used when the sender reinitiates the flush process after a repair transmission.

```
  crl [flushProcess] :
  {< A : SenderApplication | dataBuffer : ML >
    < S : DTsender |
      children : OS, NormRobustFactor : NZN, GRTT : T,
      dataBuffer : M, flushBuffer : M',
      FLUSHcounter : N, FLUSHtimer : 0,
      NACKaccumTimer : INF, repairTransmission : nil >
    OC:ObjectConfiguration}
    =>
  {< A : SenderApplication | >
    < S : DTsender |
      flushBuffer : (if s(N) == NZN then noObjectBlock else M' fi),
      FLUSHcounter : (if s(N) == NZN then 0 else s(N) fi),
      FLUSHtimer : (if s(N) == NZN then INF else 2 * T fi) >
    (multisend FLUSH(lastDataUnitId(M'), T,
      (if ML == nil and M == noObjectBlock then true else false fi))
     from S to OS)
    OC:ObjectConfiguration}
    if M' =/= noObjectBlock /\ N < NZN .
```

Both FLUSH rules are *global* rules that rewrite terms of sort ObjectConfiguration. This is done to ensure that the order of the messages from the sender is preserved. If a FLUSH message is generated immediately after a DATA message (for example when the sender has finished transmitting the segments of an object block and sends the first FLUSH), both messages will be released into the configuration simultaneously. The FLUSH rules make sure that any other messages have been picked up by the links before a FLUSH is generated.

7.6.3 The NACK Accumulation Period

If the sender receives a NACK message, it will initiate its repair process, provided the NACK content is valid, and provided it is not already in a NACK accumulation period or in a holdoff period. If the NACK contains invalid repair requests, they are added to the sender's list of invalid repair requests, which will be dealt with by the SQUELCH rule in Section 7.6.6. If the sender starts a NACK accumulation period, it cancels any pending FLUSH messages.

```
  crl [beginNACKaccumulation] :
    (outOfLink NACK(DUIL, T, B) from O to S)
    < S : DTsenderAlone |
      NormRobustFactor : NZN, GRTT : T',
      dataBuffer : M, flushBuffer : M',
      accNACKcontent : nil, NACKaccumTimer : INF,
      invalidRepairRequests : DUIL',
      repairCycleHoldoffTimer : TI >
    =>
    (if keepValidReqs(DUIL, M', M) == (nil).DataUnitIdList
     then
      *** there are no valid requests, so the sender doesn't
      *** start a NACK accumulation period
    < S : DTsenderAlone |
      invalidRepairRequests : addDUIlist(DUIL, DUIL') >
     else
     *** there are valid repair requests and the sender sets
     *** its NACKaccumTimer
    < S : DTsenderAlone |
      accNACKcontent : keepValidReqs(DUIL, M', M),
      NACKaccumTimer : (NZN + 1) * T',
      invalidRepairRequests :
        addDUIlist(keepInvalidReqs(DUIL, M', M), DUIL'),
      repairCycleHoldoffTimer : INF,
      FLUSHcounter : 0, FLUSHtimer : INF > fi)
    if TI == INF or TI == 0
    /\ not(M == noObjectBlock and M' == noObjectBlock) .
```

During the NACK accumulation timeout, the sender continues its transmission of enqueued data content. Any arriving repair requests are dealt with as in the previous rule.

```
 crl [NACKaccumulationPeriod] :
   (outOfLink NACK(DUIL, T, B) from O to S)
   < S : DTsenderAlone |
     dataBuffer : M, flushBuffer : M',
     accNACKcontent : DUIL', NACKaccumTimer : TI,
     invalidRepairRequests : DUIL'' >
   =>
   < S : DTsenderAlone |
     accNACKcontent : addDUIlist(keepValidReqs(DUIL, M', M), DUIL'),
     invalidRepairRequests :
       addDUIlist(keepInvalidReqs(DUIL, M', M), DUIL'') >
   if TI =/= INF /\ TI =/= 0 .
```

7.6.4 Repair Transmission

When the NACK accumulation timer expires, the sender stops its ordinary transmission and transmits a repair message for every segment requested. The first rule prepares the repair transmission by moving the requests from accNACKcontent to repairTransmission and setting the data transmission timer, if it is not already running. At the same time, the holdoff timer is set.

```
 rl [endOfNACKaccumulation] :
   < S : DTsender |
     GRTT : T, dataBuffer : M, flushBuffer : M',
     repairTransmission : DUIL, accNACKcontent : DUIL',
     NACKaccumTimer : 0, repairCycleHoldoffTimer : INF,
     sendRate : T', dataTransTimer : TI >
```

```
    =>
    < S : DTsender |
      repairTransmission :
        addDUIlist(replaceObjReq(DUIL', M, M'), DUIL),
      accNACKcontent : nil,
      NACKaccumTimer : INF, repairCycleHoldoffTimer : T,
      *** If TI = INF, then TI is set to the send rate,
      *** otherwise we keep the current value of TI:
      dataTransTimer : min(T', TI) > .
```

The second rule models the actual transmission of repair DATA messages. If there is content in the flushBuffer, the flush process was probably interrupted by the start of the repair process, in which case the sender reinitiates the flushing after the last repair message has been sent. Once the repair transmission is over, it can resume its transmission of new data content.

```
  rl [repairTransmission] :
    < A : SenderApplication | dataBuffer : ML >
    < S : DTsender |
      children : OS, GRTT : T, sendRate : T',
      dataBuffer : M, flushBuffer : M',
      repairTransmission : DUI ; DUIL,
      NormRobustFactor : NZN, dataTransTimer : 0, FLUSHtimer : INF >
      =>
    < A : SenderApplication | >
    < S : DTsender |
      currentTransPos : DUI, lastRepairDataId : DUI,
      repairTransmission : DUIL,
      dataTransTimer :
        (if ML == nil and allSegsTransmitted(M) and DUIL == nil
         then INF else T' fi),
      FLUSHtimer :
        (if M' =/= noObjectBlock and DUIL == nil then 0 else INF fi) >
    (multisend DATA(DUI, noOfSegs(DUI, M', M), T, true) from S to OS) .
```

7.6.5 Holdoff Timeout After a NACK Accumulation Period

After a NACK accumulation period, the sender observes a holdoff period of one GRTT. If a NACK message arrives during the holdoff timeout, the sender will include the repair request in its ongoing repair transmission only if the data content requested has a smaller data unit identifier than the last repair message sent. If not, the request is ignored. Any invalid NACK content is put in the invalidRepairRequests list.

```
  crl [NACKarrivesDuringHoldoffTimeout] :
    (outOfLink NACK(DUIL, T, B) from O to S)
    < S : DTsenderAlone |
      currentTransPos : DUI,
      dataBuffer : M, flushBuffer : M', repairTransmission : DUIL',
      repairCycleHoldoffTimer : TI, invalidRepairRequests : DUIL'' >
    =>
    < S : DTsenderAlone |
      repairTransmission :
        addDUIlist(keepForImmediateRepair(DUIL, DUI, M', M), DUIL'),
      invalidRepairRequests :
        addDUIlist(keepInvalidReqs(DUIL, M', M), DUIL'') >
    if TI =/= INF and TI =/= 0 .
```

The following rule simply turns the holdoff timer off when it expires. The sender is then free to start a new NACK accumulation period.

```
  rl [repairCycleHoldoffTimerExpires] :
    < S : DTsender |
      repairCycleHoldoffTimer : 0 >
    =>
    < S : DTsender |
      repairCycleHoldoffTimer : INF > .
```

7.6.6 Notifying the Receiver of Invalid Repair Requests

When the sender receives invalid repair requests from a receiver, i.e., requests for data content it no longer buffers, it multicasts a SQUELCH message to advertise the earliest valid repair position. Transmission of SQUELCH messages is limited to once per $2 * GRTT$.

```
  crl [sendSQUELCH] :
    < S : DTsender |
      children : OS, GRTT : T,
      dataBuffer : M, flushBuffer : M',
      invalidRepairRequests : DUIL, SQUELCHholdoffTimer : TI >
    =>
    < S : DTsender |
      invalidRepairRequests : nil, SQUELCHholdoffTimer : 2 * T >
    (if M' == noObjectBlock
     then (multisend SQUELCH(firstDataUnitId(M), T) from S to OS)
     else (multisend SQUELCH(firstDataUnitId(M'), T) from S to OS)
     fi)
    if DUIL =/= nil /\ (TI == 0 or TI == INF) .
```

When the SQUELCHholdoffTimer expires, it is turned off, and the sender is free to send a new SQUELCH for invalid repair requests.

```
rl [SQUELCHholdoffTimerExpires] :
  < S : DTsender | SQUELCHholdoffTimer : 0 >
  =>
  < S : DTsender | SQUELCHholdoffTimer : INF > .
```

7.7 The DTreceiver Rules

This section contains the rewrite rules for the receiver's actions. First I specify reception of data content, initiation of a NACK cycle at an object boundary, reception of repair messages, and forwarding of data content from a receiver to its application. I then model how a NACK cycle is initiated by a FLUSH message, and the NACK holdoff timeout and NACK generation. Finally, I specify how the receiver reacts to a SQUELCH message from the sender.

7.7.1 Reception of Data Content and Initiation of NACK Cycle At Object Boundary

In the following rule, the receiver receives the next expected segment. The nextExpectedDUI value signifies that all previous segments have been received (or at least have been accounted for), and as a consequence the receiver does not need to check for repair needs when it reaches an object boundary. The receiver knows that the identification of objects and segments is ordered, so the updated nextExpectedDUI value will be either the next data unit identifier of the current object, or the first id of the next object.

```
rl [receivesTheNextExpectedSegment] :
  (outOfLink DATA(NZN :: NZN', NZN'', T, false) from O to R)
  < R : DTreceiver |
    receiveBuffer : ML, nextExpectedDUI : NZN :: NZN' >
  =>
  < R : DTreceiver |
    GRTT : T,
    receiveBuffer : ML ++ DATA(NZN :: NZN', NZN'', T, false),
    sendersCurrTransPos : NZN :: NZN',
    nextExpectedDUI : (if NZN' < NZN'' then (NZN :: s(NZN'))
                        else (s(NZN) :: 1) fi) > .
```

As opposed to the above case, receiving a segment with a data unit identifier other than the one expected implies there is a gap in reception (since the network is specified so that messages are delivered in order, it cannot be the case that a new segment with a higher identifier has arrived before one with a lower identifier). If it has crossed an object boundary, the receiver sets its NACK backoff timer, provided it is not already in a NACK cycle or in a holdoff period. It also places information about the candidate NACK content in the repairRequests attribute. Regardless of whether it starts a NACK cycle or not, it makes a note of which segments are missing in repairNeeds, using the information in nextExpectedDUI and receiveBuffer.

```
crl [gapInReception] :
  (outOfLink DATA(DUI, NZN, T, false) from O to R)
  < R : DTreceiver |
    randomSeed : N, NormRobustFactor : NZN', gsize : NZN'',
    receiveBuffer : ML, nextExpectedDUI : DUI',
    repairNeeds : DUIL, repairRequests : DUIL',
    NACKbackoffTimer : TI, NACKcycleHoldoffTimer : TI' >
  =>
  (if objectBoundary(DUI, ML)
   and TI == INF and (TI' == 0 or TI' == INF)
   then *** NACK cycle is triggered at an object boundary
  < R : DTreceiver |
    randomSeed : random(N), GRTT : T,
    receiveBuffer : ML ++ DATA(DUI, NZN, T, false),
    sendersCurrTransPos : DUI,
    repairNeeds : DUIL ; recordRepairNeeds(DUI, DUI', ML),
    repairRequests :
      addDUIlist(DUIL,
                 (DUIL' ; recordRepairNeeds(DUI, DUI', ML))),
    NACKbackoffTimer : randomBackoff(random(N), NZN', T, NZN'') >
   else *** the seg. and info. on missing segments is simply stored
  < R : DTreceiver |
    GRTT : T,
    receiveBuffer : (if largerThan(DUI, DUI')
                     then ML ++ DATA(DUI, NZN, T, false)
                     else ML fi), *** duplicates are ignored
    sendersCurrTransPos : DUI,
    repairNeeds : DUIL ; recordRepairNeeds(DUI, DUI', ML) > fi)
  if DUI =/= DUI' .
```

7.7.2 Reception of Repair Messages

When the receiver gets a repair segment (in a DATA message with the repair flag set), it includes the segment in its receive buffer. If the repair segment covers up a gap in the buffer, the receiver updates the nextExpectedDUI attribute. Any pending repair requests for the segment are cancelled. If the repair message carries a segment it has already received, the receiver ignores the message.

```
rl [receivesRepairSegment] :
  (outOfLink DATA(DUI, NZN, T, true) from O to R)
  < R : DTreceiver |
    receiveBuffer : ML, nextExpectedDUI : DUI',
    repairNeeds : DUIL, repairRequests : DUIL' >
  =>
  if missingSeg(DUI, DUIL) *** this is a missing segment
```

```
then
< R : DTreceiver |
  GRTT : T, sendersCurrTransPos : DUI,
  receiveBuffer : addRepairMsg(DATA(DUI, NZN, T, true), ML),
  repairNeeds : removeRepReq(DATA(DUI, NZN, T, true), DUIL),
  repairRequests : removeRepReq(DATA(DUI, NZN, T, true), DUIL'),
  nextExpectedDUI :
  (if DUI == DUI' then
    newNextExpectedDUI(
      recBuffFromDUI(addRepairMsg(DATA(DUI, NZN, T, true), ML), DUI))
   else DUI' fi) >
else *** the message is ignored
< R : DTreceiver | GRTT : T > fi .
```

7.7.3 Forwarding Data Content to the Application

In the following rule, the receiver forwards every data segment, except the last segment received, to its application. The DATA message carrying this segment provides information about the number of segments in the object, so the receiver keeps it in case it needs to generate a repair request if the next expected segment fails to arrive.

```
crl [forwardDataSegments] :
  < A : ReceiverApplication | receiver : R, dataBuffer : DUIL >
  < R : DTreceiver |
    receiveBuffer : DATA(NZN :: NZN', NZN'', T, B) ++ ML,
    nextExpectedDUI : N :: N' >
  =>
  < A : ReceiverApplication | dataBuffer : DUIL ; (NZN :: NZN') >
  < R : DTreceiver | receiveBuffer : ML >
  if (not(NZN' == NZN'' and N' == 1 and s(NZN) == N)
    and not(NZN :: s(NZN') == N :: N'))
  or N :: N' == (0 :: 0) .
```

When the receiver has received the very last segment (i.e., when neither the sender application nor the NORM sender have any more data to send), it forwards the segment to its application and sets the nextExpectedDUI attribute to 0 :: 0:

```
rl [forwardLastDataSegment] :
  < A : ReceiverApplication | receiver : R, dataBuffer : DUIL >
  < R : DTreceiver | receiveBuffer : DATA(NZN :: NZN', NZN', T, B) >
  < S : DTsender | dataBuffer : noObjectBlock, flushBuffer : noObjectBlock >
  < A' : SenderApplication | dataBuffer : (nil).MsgList >
  =>
  < A : ReceiverApplication | dataBuffer : DUIL ; (NZN :: NZN') >
  < R : DTreceiver |
    receiveBuffer : (nil).MsgList, nextExpectedDUI : 0 :: 0 >
  < S : DTsender | > < A' : SenderApplication | > .
```

7.7.4 NACK Cycle Initiated by FLUSH

When a receiver gets a FLUSH message, it will initiate a NACK cycle, provided it has repair needs in the range up to and including the data unit identifier in the message, and provided its backoff and holdoff timers are not running.

```
crl [NACKcycleInitiatedByFLUSH] :
  (outOfLink FLUSH(NZN :: NZN', T, B) from O to R)
  < R : DTreceiver |
    randomSeed : N, NormRobustFactor : NZN'', gsize : NZN''',
    receiveBuffer : ML, nextExpectedDUI : DUI,
    repairNeeds : DUIL, repairRequests : DUIL',
    NACKbackoffTimer : TI, NACKcycleHoldoffTimer : TI',
    sendersCurrTransPos : DUI' >
  =>
  (if    *** the receiver has valid repair requests,
    (keepValidNACKcontent(DUIL, NZN :: NZN') =/= nil
     or recordRepairNeeds(s(NZN) :: 1, DUI,
                         recBuffUpToDUI(ML, s(NZN) :: 1)) =/= nil)
         *** and its backoff and holdoff timers are off
    and TI == INF and (TI' == INF or TI' == 0)
    then  *** initiate NACK cycle
    < R : DTreceiver |
      GRTT : T, randomSeed : random(N),
      repairNeeds : addDUIlist(recordRepairNeeds(s(NZN) :: 1, DUI,
                                   recBuffUpToDUI(ML, s(NZN) :: 1)),
                               DUIL),
      repairRequests : addDUIlist(recordRepairNeeds(s(NZN) :: 1, DUI,
                                     recBuffUpToDUI(ML, s(NZN) :: 1)),
                                 keepValidNACKcontent(DUIL, NZN :: NZN')),
      NACKbackoffTimer : randomBackoff(random(N), NZN'', T, NZN'''),
      sendersCurrTransPos : (if B then 0 :: 0 else DUI' fi) >
    else  *** just update next expected data unit id
    < R : DTreceiver |
      GRTT : T,
      nextExpectedDUI :
        (if B == true and DUI == s(NZN) :: 1 then 0 :: 0 else DUI fi) > fi)
  if largerThan(s(NZN) :: 1, DUI) /\ DUI =/= 0 :: 0 .
```

If the FLUSH concerns data it has already received, the receiver simply ignores the message.

```
crl [ignoreFLUSH] :
  (outOfLink FLUSH(DUI, T, B) from O to R)
  < R : DTreceiver | nextExpectedDUI : DUI' >
  =>
  < R : DTreceiver | GRTT : T >
  if smallerThan(DUI, DUI') or DUI' == 0 :: 0 .
```

7.7.5 External Repair State Accumulation

While its backoff timer is running, the receiver records the external repair state as it receives NACK messages from other receivers. If it is not in a NACK accumulation period, any external repair requests are ignored.

```
rl [accumulateExternalRepairState] :
  (outOfLink NACK(DUIL, T, B) from O to R)
  < R : DTreceiver |
    extRepRequests : DUIL', NACKbackoffTimer : TI >
  =>
  if TI =/= INF and TI =/= 0
  then
  < R : DTreceiver |
    extRepRequests : addDUIlist(DUIL, DUIL') >
  else
  < R : DTreceiver | > fi .
```

7.7.6 Transmission of NACK Followed by Holdoff Timeout

When the backoff timer expires, the receiver makes a decision on whether to send a NACK. If it has repair needs with data unit identifiers smaller than the sender's current transmission position (as recorded by the receiver) and these repair needs are not covered by requests from other receivers, the receiver generates a NACK message. The NACK contains the subset of the repair requests recorded at the time of the NACK cycle initiation starting with the earliest repair position up to the sender's current transmission position. At the same time, it sets the holdoff timer which prevents it from immediately initiating a new NACK cycle.

```
crl [makeNACKdecision] :
  < R : DTreceiverAlone |
    parent : O, NormRobustFactor : NZN, GRTT : T,
    CLR : B, sendersCurrTransPos : DUI,
    repairRequests : DUIL, extRepRequests : DUIL',
    NACKbackoffTimer : 0, NACKcycleHoldoffTimer : INF >
  =>
  (if not(alreadyNACKed(DUIL, DUIL')) and
    (DUI == 0 :: 0 or-else smallerThan(first(DUIL), DUI))
```

```
     then
     (< R : DTreceiverAlone |
       repairRequests : nil, extRepRequests : nil,
       NACKbackoffTimer : INF, NACKcycleHoldoffTimer : (NZN + 2) * T >
     (intoLink NACK(sublistUpToDUI(DUIL, DUI), 0, B) from R to O))
     else
     < R : DTreceiverAlone |
      repairRequests : nil, extRepRequests : nil,
      NACKbackoffTimer : INF > fi)
    if DUIL =/= nil .
```

During the backoff timeout, the receiver may have received repair messages or a SQUELCH for its missing segments and removed the corresponding requests from repairRequests—if so, the backoff timer is turned off once it expires.

```
  rl [noRepairRequests] :
    < R : DTreceiver | NACKbackoffTimer : 0, repairRequests : nil >
    =>
    < R : DTreceiver | NACKbackoffTimer : INF > .
```

When the holdoff timeout expires, the timer is turned off and the receiver is free to start a new repair request cycle at the next NACK cycle initiation event.

```
  rl [turnOffNACKcycleHoldoffTimer] :
    < R : DTreceiver | NACKcycleHoldoffTimer : 0 >
    =>
    < R : DTreceiver | NACKcycleHoldoffTimer : INF > .
```

7.7.7 Cancel Pending Invalid Repair Requests

If it receives a SQUELCH message, the receiver updates its lists of repair needs and pending repair requests. Any records of missing segments that are invalidated by the SQUELCH are removed.

```
  rl [removeInvalidRepairRequests] :
    (outOfLink SQUELCH(DUI, T) from O to R)
    < R : DTreceiver |
      receiveBuffer : ML, nextExpectedDUI : DUI',
      repairNeeds : DUIL, repairRequests : DUIL' >
    =>
    < R : DTreceiver |
      GRTT : T, repairNeeds : removeSquelchedRepReqs(DUIL, DUI),
      repairRequests : removeSquelchedRepReqs(DUIL', DUI),
      nextExpectedDUI : (if smallerThan(DUI', DUI)
        then newNextExpectedDUI(recBuffFromDUI(ML, DUI))
        else DUI' fi) > .
```

7.8 delta and mte for the DTreceiver and DTsender Classes

As in the Real-Time Maude specification of the RTT component, the functions `delta` and `mte` must be defined for `DTsenderAlone` and `DTreceiverAlone` objects:

```
  eq delta(< S : DTsenderAlone | clock : T, dataTransTimer : TI,
                  NACKaccumTimer : TI', repairCycleHoldoffTimer : TI'',
                  FLUSHtimer : TI''' >, T') =
         < S : DTsenderAlone | clock : T + T', dataTransTimer : TI monus T',
                  NACKaccumTimer : TI' monus T',
                  repairCycleHoldoffTimer : TI'' monus T',
                  FLUSHtimer : TI''' monus T' > .

  eq delta(< R : DTreceiverAlone | clock : T, NACKbackoffTimer : TI,
                  NACKcycleHoldoffTimer : TI' >, T') =
         < R : DTreceiverAlone | clock : T + T',
                  NACKbackoffTimer : TI monus T',
                  NACKcycleHoldoffTimer : TI' monus T' > .


  eq mte(< R : DTreceiverAlone | NACKbackoffTimer : TI,
                  NACKcycleHoldoffTimer : TI' >) =
       min(TI, TI') .

  eq mte(< S : DTsenderAlone | dataTransTimer : TI, NACKaccumTimer : TI',
                  repairCycleHoldoffTimer : TI'', FLUSHtimer : TI''' >) =
       min(TI, min(TI', min(TI'', TI'''))) .
```

Chapter 8

Analysing the Data and Repair Transmission Component

In this chapter, I formally analyse the Real-Time Maude specification of the data and repair transmission component of the NORM protocol. The goal of the component is for the sender to reliably transmit one or more data objects to a group of receivers on behalf of an application program. If network errors occur, the component uses nonacknowledgment techniques to retransmit lost segments of data. The protocol is highly nondeterministic, and consequently there is a combinatorial explosion in the state space. Because of this, and because my computer does not have sufficient random access memory, it became impossible to execute some of my analysis commands within reasonable time. However, what I was able to analyse illustrates my comments in Chapter 7 about certain parts of the component that appear not to be fully specified, namely the receiver initiation of a repair request, and the sender's FLUSH process.

For the analysis, I define two initial states, which are similar except for the delay in the router. The states have the same topology as the initial state `rtt1` in Chapter 6, with the addition of an application layer.

8.1 A Simple Initial State: `data1`

In the module `NORM-DT-1`, which imports the `NORM-DATA-TRANSMISSION` module, I define the initial state `data1`. The state has three NORM protocol objects: one sender and two receivers. Each of these sends or receives data on behalf of an application object. The sender and receivers are connected by a router and three link objects. The topology of `data1`, which resembles that of the state `rtt1` in Chapter 6, is shown in Figure 8.1.

```
  op data1 : -> GlobalSystem .
  eq data1 =
 {< senderApp : SenderApplication | dataBuffer :
         objectBlock(OBJECT(1, 70, 0) ++ OBJECT(2, 70, 0))
```

```
          ++  objectBlock(OBJECT(3, 70, 0) ++ OBJECT(4, 70, 0)) >
< sender : DTsenderAlone | clock : 0, NormRobustFactor : 4, GRTT : 70,
       gsize : 2, children : router, sendRate : 10,
       dataBuffer : noObjectBlock, flushBuffer : noObjectBlock,
       repairTransmission : nil, currentTransPos : 0 :: 0,
       lastNewDataId : 0 :: 0, lastRepairDataId : 0 :: 0,
       dataTransTimer : INF, accNACKcontent : nil,
       invalidRepairRequests : nil, NACKaccumTimer : INF,
       repairCycleHoldoffTimer : INF, FLUSHtimer : INF,
       FLUSHcounter : 0, SQUELCHholdoffTimer : INF >
< sender-router : Link | upNode : sender, downNode : router,
       propDelay : 5, bandwidth : 10, LINK-ATTS >
< router : Router | parent : sender, children : rec1 rec2,
       buffer : (nil).MsgList, bufferCap : 2, queuingDelay : 12 >
< router-rec1 : Link | upNode : router, downNode : rec1,
       propDelay : 11, bandwidth : 1, LINK-ATTS >
< router-rec2 : Link | upNode : router, downNode : rec2,
       propDelay : 15, bandwidth : 1, LINK-ATTS >
< rec1 : DTreceiverAlone | randomSeed : 77, CLR : false, RCV-ATTS >
< rec2 : DTreceiverAlone | randomSeed : 23, CLR : true, RCV-ATTS >
< recApp1 : ReceiverApplication |
       receiver : rec1, dataBuffer : (nil).DataUnitIdList >
< recApp2 : ReceiverApplication |
       receiver : rec2, dataBuffer : (nil).DataUnitIdList >} .
```

In the above state, *LINK-ATTS* denotes the following link attributes

```
upstream : nil, downstream : nil
```

and *RCV-ATTS* the following receiver attributes

```
clock : 0, NormRobustFactor : 4, GRTT : 0, gsize : 2, parent : router,
receiveBuffer : nil, nextExpectedDUI : 1 :: 1, sendersCurrTransPos : 0 :: 0,
repairNeeds : nil, repairRequests : nil, extRepRequests : nil,
NACKbackoffTimer : INF, NACKcycleHoldoffTimer : INF
```

There are two object blocks in the sender application's data buffer, each containing two objects to be transmitted. The objects are numbered 1 to 4, and each object is made up of 70 segments. The sender's GRTT estimate is set to a fixed value, 70, based on the RTT value of the current limiting receiver, `rec2`.

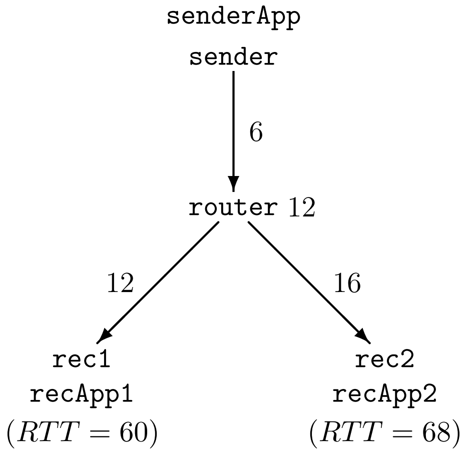

Figure 8.1: The topology corresponding to the initial state data1

8.2 Analysing *One* Possible Behaviour of the Data Transmission Component from State data1

The first step in our analysis is to simulate one of the possible behaviours of the data transmission component from state data1 with the timed rewrite command. The NORM specification does not put a limit on the number of times the flush and repair processes can be reinitiated, and in theory the repair messages could be dropped and the receivers forced to ask for repair again and again. However, this is an unlikely scenario in this model, because there is no other traffic in the network than that of the NORM sender and receivers. Therefore, the rewrite command is executed without a time bound. When simulating a behaviour of the component, I expect that the execution will lead to a state where the receiver applications have received every data segment transmitted by the NORM sender:

```
Maude> (trew data1 with no time limit .)

Result ClockedSystem :
 {< senderApp : SenderApplication | dataBuffer :(nil).MsgList >
  < sender : DTsenderAlone | FLUSHcounter : 0, FLUSHtimer : INF, GRTT : 70,
        NACKaccumTimer : INF, NormRobustFactor : 4,
        SQUELCHholdoffTimer : INF, accNACKcontent : nil, children : router,
        clock : 4240, currentTransPos : 4 :: 60, dataBuffer : noObjectBlock,
        dataTransTimer : INF, flushBuffer : noObjectBlock, gsize : 2,
        invalidRepairRequests : nil, lastNewDataId : 4 :: 70,
        lastRepairDataId : 4 :: 60, repairCycleHoldoffTimer : INF,
        repairTransmission : nil, sendRate : 10 >
  < sender-router : Link | bandwidth : 10, downNode : router, downstream : nil,
        propDelay : 5, upNode : sender, upstream : nil >
  < router : Router | bufferCap : 2, buffer : nil, children :(rec1 rec2),
        parent : sender, queuingDelay : 12 >
  < router-rec1 : Link | bandwidth : 1, downNode : rec1, downstream : nil,
        propDelay : 11, upNode : router, upstream : nil >
< router-rec2 : Link | bandwidth : 1, downNode : rec2, downstream : nil,
        propDelay : 15, upNode : router, upstream : nil >
< rec1 : DTreceiverAlone | CLR : false, GRTT : 70, NACKbackoffTimer : INF,
        NACKcycleHoldoffTimer : INF, NormRobustFactor : 4, clock : 4240,
        extRepRequests : nil, gsize : 2, nextExpectedDUI : 0 :: 0,
        parent : router, randomSeed : 2107, receiveBuffer : nil,
        repairNeeds : nil, repairRequests : nil,
        sendersCurrTransPos : 4 :: 60 >
< rec2 : DTreceiverAlone | CLR : true, GRTT : 70, NACKbackoffTimer : INF,
        NACKcycleHoldoffTimer : INF, NormRobustFactor : 4, clock : 4240,
        extRepRequests : nil, gsize : 2, nextExpectedDUI : 0 :: 0,
        parent : router, randomSeed : 6585, receiveBuffer : nil,
        repairNeeds : nil, repairRequests : nil,
        sendersCurrTransPos : 4 :: 60 >
< recApp1 : ReceiverApplication | dataBuffer :((1 :: 1);(1 :: 2);(1 :: 3);
        (1 :: 4);(1 :: 5);(1 :: 6);(1 :: 7); [...] ;(4 :: 63);(4 :: 64);
        (4 :: 65);(4 :: 66);(4 :: 67);(4 :: 68);(4 :: 69); 4 :: 70),
        receiver : rec1 >
< recApp2 : ReceiverApplication | dataBuffer :((1 :: 1);(1 :: 2);(1 :: 3);
        (1 :: 4);(1 :: 5);(1 :: 6);(1 :: 7); [...] ;(4 :: 63);(4 :: 64);
        (4 :: 65);(4 :: 66);(4 :: 67);(4 :: 68);(4 :: 69); 4 :: 70),
        receiver : rec2 >} in time 4240
```

Indeed, the rewrite command produces a state, reachable in time 4240, where the transmission has terminated successfully. Both `recApp1` and `recApp2` have received every segment of the four objects sent, 280 segments in all. By looking at the sender's attributes, we see that there has been at least one case of packet loss. The sender attribute `lastNewDataId` contains the identifier of the last *new* data segment sent, namely segment 4 :: 70. The attribute `lastRepairDataId` has the value 4 :: 60, which means that segment 4 :: 60 was lost and a repair segment was sent. Tracing the execution with the Real-Time Maude trace commands shows that *seven* data segments were lost and successfully repaired by the sender in this behaviour of the component.

8.3 Model Checking the Data Transmission Component from State `data1`

In this section, I set out to analyse further the data transmission component from the initial state `data1`, by searching for an undesired behaviour and by model checking the component. The analysis effort runs into trouble when it turns out that the computer I am using has insufficient memory, and I am only able to perform model checking within a limited time interval.

Although rewriting the state `data1` in the data transmission specification resulted in a state where transmission was successfully completed, the rewrite command simulates just *one* of the possible behaviours of a system. We cannot tell from just rewriting the specification whether there is a behaviour where one or both receivers did *not* get all the segments. To see if there is such an undesired behaviour, we can for instance search for a state reachable from the initial state where the sender has completed transmisssion (that is, it has no data in its buffers), and a repair request is delivered to the sender. Such a state could not be further rewritten, because once the sender has finished transmission, it does not respond to messages from the receivers any more. The following timed search command checks if a state where the sender receives a repair request but has already finished transmission is reachable from the initial state within time 5000:

```
Maude> (tsearch [1] data1 =>*
            {< sender : DTsenderAlone | dataBuffer : noObjectBlock,
                          flushBuffer : noObjectBlock, ATTS:AttributeSet >
             (outOfLink NACK(DL:DataUnitIdList, T:Time, B:Bool)
                          from router to sender)
             REST:Configuration}
        in time < 5000 .)

Abort
```

After a while, the search was aborted. This is not really a surprising result. There is a great deal of nondeterminism in the specification, leading to a combinatorial explosion in the state space. For instance, every time the router forwards a message from a node, it releases n copies of the message into the configuration simultaneously for n outgoing links. Any of the messages can be picked up first, so there are at least n possible successors to this state. In addition, other actions, such as a receiver forwarding segments to its application, may happen concurrently, thus increasing the number of possible successor states. In the rewrite from the initial state `data1` in the previous section, it took 5233 rewrite steps to reach the result at time 4240. The Maude search tool, which the Real-Time Maude search command is built on, performs breadth-first search. In order to avoid searching from states it has already visited, the tool caches *every* state is sees in a search. Although the Maude search command is very efficiently implemented, the above search clearly exhausted the memory of the computer I use.

I tried several ways of getting around this problem. First, I tried to use the search command to find a state where a receiver had experienced packet loss, in order to search from this state to see if there was a behaviour where the packet loss had *not* been repaired by the time the sender had completed transmission. However, the search was aborted after a while. I then looked at the trace of the behaviour from Section 8.2, found a state where both receivers had repair needs, and performed a search from this state. Again, the search failed. The trace from the previous section reveals that the first state in the behaviour where a receiver becomes aware that a data packet is missing, occurs at time 1723: `rec1` receives a packet with sequence number (3 :: 2) and discovers that it has not received (3 :: 1). I am not able to check whether this is the first case of packet loss in *every* behaviour from the initial state, but the following search shows that at least within time 350 neither of the receivers have experienced any packet loss. The search command below searches for a state, reachable from the state `data1` within time 350, where a receiver has recorded a missing data segment in its `repairNeeds` attribute:

```
Maude> (tsearch data1 =>*
               {< O:Oid : DTreceiverAlone |
                     repairNeeds : (NZN:NzNat :: N:Nat) ; DUIL:DataUnitIdList,
                     ATTS:AttributeSet >
                REST:Configuration}
        in time < 350 .)

No solution
```

Will we be more successful in using the model checking command? The property I am most interested in checking is whether it is always the case that if a data packet is sent, eventually it will reach both receivers. This can be expressed in linear time temporal logic with a formula $\Box(packetIsSent(x) \rightarrow (\Diamond\, rec1OK(x) \wedge \Diamond\, rec2OK(x)))$, where x is a sequence number, and where $packetIsSent(x)$ means that the sender has sent a data packet with sequence number x, and $rec1OK(x)$ and $rec2OK(x)$ means that receiver 1 and receiver 2 have received this packet. The three atomic propositions are defined in the module MODEL-CHECK-NORM-DT-1 as packetIsSent, rec1OK and rec2OK:

```
(tomod MODEL-CHECK-NORM-DT-1 is
  including TIMED-MODEL-CHECKER .
  protecting NORM-DT-1 .

  vars DUI DUI' : DataUnitId .   var NZN : NzNat .   var T : Time .
  var REST : Configuration .     var B : Bool .      vars ML ML' : MsgList .

  op packetIsSent : DataUnitId -> Prop [ctor] .
  op rec1OK : DataUnitId -> Prop [ctor] .
  op rec2OK : DataUnitId -> Prop [ctor] .
```

The atomic proposition packetIsSent(*segId*) holds in every state where the sender has just released a DATA message with segment identifier *segId*:

```
  eq {(intoLink DATA(DUI, NZN, T, B) from sender to router) REST}
          |= packetIsSent(DUI') = DUI == DUI' .
```

The atomic propositions rec1OK(*segId*) and rec2OK(*segId*) hold in every state where rec1 and rec2, respectively, have received a DATA message with segment identifier *segId* (either the original, or a repair message). This is shown by the value of the sendersCurrTransPos attribute:

```
  eq {< rec1 : DTreceiverAlone | sendersCurrTransPos : DUI >  REST}
          |= rec1OK(DUI') = DUI == DUI' .

  eq {< rec2 : DTreceiverAlone | sendersCurrTransPos : DUI >  REST}
          |= rec2OK(DUI') = DUI == DUI' .
endtom)
```

To find out whether it is always the case in the component w.r.t. initial state `data1` that both receivers sooner or later receive a data segment transmitted by the sender, we would have to check the above formula for *every* data segment in the initial state. In order to check this property for the 280 segments of `data1` I will have to execute the model checking command with a large time bound. A few attempts at using the command with a large time bound fail, because of the size of the state space. However, I can restrict the model checking to examining if, say, the first *ten* segments are received by both receivers. I first try to find a suitable time bound in order to delimit the model checking. The `find latest` command searches through the possible behaviours from the initial state, looking for the behaviour where it takes the *longest* time before a receiver's `sendersCurrTransPos` attribute has the value (1 :: 10). It returns the *first* occurence of such a state in that behaviour:

```
Maude> (find latest data1 =>*
           {< O:Oid : DTreceiverAlone | sendersCurrTransPos : 1 :: 10,
                                        ATTS:AttributeSet >
            REST:Configuration}
        with no time limit .)

Result:
 {< rec1 : DTreceiverAlone | [...], sendersCurrTransPos : 1 :: 10 >
  < rec2 : DTreceiverAlone | [...], sendersCurrTransPos : 1 :: 9 >
  [...]} in time 150
```

In the state returned by the command, reachable from the initial state at time 150, `rec1` has just received segment number (1 :: 10). Thus, checking if messages (1 :: 1) through (1 :: 10) are reliably transmitted within time 200 seems like a suitable delimitation. The `find latest` command, because it looks for a match with the search pattern in *every* possible behaviour from the initial state, and returns a negative answer if no match is found, is a model checking command for *eventually* properties. The result from the command above shows that eventually, within the given time interval, some receiver will get data segment (1 :: 10) in every possible behaviour. However, I would like to check that *both* receivers will eventually get segments (1 :: 1) through (1 :: 10). The model checking commands below check if, in every behaviour of the component w.r.t. the initial state `data1`, sooner or later data segment s is sent, and that if s is sent, sooner or later both receivers will get it[1]:

```
Maude> (mc data1 |=t
           (<> packetIsSent(1 :: 1)) /\
           (packetIsSent(1 :: 1) => (<> rec1OK(1 :: 1) /\ <> rec2OK(1 :: 1)))
        in time <= 200 .)

Result Bool : true
```

[1] Again, I use `A => B`, which is defined as `[] (A -> B)` in the Maude model checker [14].

```
Maude> (mc data1 |=t
           (<> packetIsSent(1 :: 2)) /\
           (packetIsSent(1 :: 2) => (<> rec1OK(1 :: 2) /\ <> rec2OK(1 :: 2)))
        in time <= 200 .)

Result Bool : true

   .
   .
   .

Maude> (mc data1 |=t
           (<> packetIsSent(1 :: 10)) /\
           (packetIsSent(1 :: 10) => (<> rec1OK(1 :: 10) /\ <> rec2OK(1 :: 10)))
        in time <= 200 .)

Result Bool : true
```

The sequence of model checking commands shows that it holds for the component w.r.t. the initial state within time 200 that segments (1 :: 1) through (1 :: 10) are sent, and that it is always the case that if one of these segments are sent, then eventually it will reach both receivers. Even if I had been able to verify this property for every data segment in the initial state, it would not have been a proof that the data and repair transmission component is reliable. If we were to verify that the component is reliable, we would have to check this property for *every* possible initial state. The size of the problem often makes it impossible to verify an entire system. However, model checking can be very useful for finding errors. If the property does not hold, the model checker returns a counterexample, which can be used for locating and correcting errors in the protocol.

The Maude model checker has a performance that is comparable with state-of-the-arts explicit state model checkers such as SPIN [15]. The number of reachable states causes the model checker to abort when it is used to check a property of the data transmission specification within a large time interval. However, even though we did not manage to analyse the entire problem, we could still analyse quite a lot of it.

8.4 Increasing the Network Delay: `data2`

Rewriting the state `data1` did not reveal any errors in the specification. It turned out to be difficult to perform model checking analysis because of the size of the state space, but analysing the component with the rewriting command also produces valuable results. In the rest of this chapter I will use the rewrite command to analyse a behaviour of the data and repair transmission component from a slightly different initial state. By redefining the state `data1` I hope to provoke an erroneous behaviour which I can study further. The state `data2`, defined in the timed module `NORM-DT-2`, is identical to the state `data1` except for the following values: the router now has a much larger queuing delay, and its buffer capacity is increased to

3. The increased delay in the router leads to a new greatest round-trip time estimate of 106 in the sender object.

```
  op data2 : -> GlobalSystem .
  eq data2 =
  {< sender : DTsenderAlone | GRTT : 106, [...] >
  < router : Router | bufferCap : 3, queuingDelay : 30, [...] >}
```

The new state represents a situation where increased traffic in the network causes a larger delay in a router. We can imagine that the router has received a lot of packets belonging to *other* transmissions, and that it takes a very long time for it to forward a packet. This may lead to increased packet loss, because the router's buffer is more likely to be filled up.

8.5 Analysing an Erroneous Behaviour of the Data Transmission Component from State data2

In this section, I analyse the result of rewriting the initial state data2. I have redefined the initial state so that there is an increased possibility that many packets will be thrown away, in order to analyse the protocol under heavy packet loss. The rewrite command returns a state where the transmission was not successfully completed, and by tracing the rewrite we can find out which events lead to this error state. I start by rewriting the state data2 with no time limit:

```
Maude> (trew data2 with no time limit .)

Result ClockedSystem :
 {< sender : DTsenderAlone | dataTransTimer : INF,
          dataBuffer : noObjectBlock, flushBuffer : noObjectBlock,
          FLUSHcounter : 0, FLUSHtimer : INF,
          currentTransPos : 4 :: 70, lastNewDataId : 4 :: 70,
          lastRepairDataId : 0 :: 0, [...] >
  < router : Router | buffer : dly(forward(sender,FLUSH(4 :: 70,106,true)),7),
          [...] >
  < rec1 : DTreceiverAlone | receiveBuffer :(DATA(2 :: 70,70, 106,false)
          ++ DATA(3 :: 2,70,106,false)++ DATA(3 :: 3,70,106,false)
          ++ DATA(3 :: 4,70,106,false)++ DATA(3 :: 5,70,106,false)
          ++ DATA(3 :: 6,70,106,false)++ DATA(3 :: 7,70,106,false)
          ++ [...] ++ DATA(4 :: 68,70,106,false)++ DATA(4 :: 69,70,106,false)
          ++ DATA(4 :: 70,70,106,false)),
          sendersCurrTransPos : 0 :: 0, nextExpectedDUI : 3 :: 1,
          repairNeeds : 3 :: 1, repairRequests : 3 :: 1, [...] >
  < rec2 : DTreceiverAlone | receiveBuffer :(DATA(2 :: 70,70, 106,false)
          ++ DATA(3 :: 2,70,106,false)++ DATA(3 :: 3,70,106,false)
          ++ DATA(3 :: 4,70,106,false)++ DATA(3 :: 5,70,106,false)
          ++ DATA(3 :: 6,70,106,false)++ DATA(3 :: 7,70,106,false)
          ++ [...] ++ DATA(4 :: 68,70,106,false)++ DATA(4 :: 69,70,106,false)
          ++ DATA(4 :: 70,70,106,false)),
          sendersCurrTransPos : 0 :: 0, nextExpectedDUI : 3 :: 1,
          repairNeeds : 3 :: 1, repairRequests : nil, [...] >
  < recApp1 : ReceiverApplication | dataBuffer :((1 :: 1);(1 :: 2);(1 :: 3);
          (1 :: 4);(1 :: 5);(1 :: 6);(1 :: 7); [...] ;(2 :: 65);(2 :: 66);
          (2 :: 67);(2 :: 68); 2 :: 69), receiver : rec1 >
  < recApp2 : ReceiverApplication | dataBuffer :((1 :: 1);(1 :: 2);(1 :: 3);
          (1 :: 4);(1 :: 5);(1 :: 6);(1 :: 7); [...] ;(2 :: 65);(2 :: 66);
          (2 :: 67);(2 :: 68); 2 :: 69), receiver : rec2 >
  outOfLink NACK(3 :: 1,0,true)from router to sender} in time 3453
```

The rewrite leads to a state, reachable in time 3453, where the sender has completed its transmission of data (I have removed some of the objects and attributes from the output). It has finished the flush process, and emptied its `dataBuffer` and `flushBuffer`. Its final `FLUSH` message is in the router's buffer, waiting to be forwarded to the receivers. However, a `NACK` message from `rec2` has just been released into the configuration by the router. Increasing the delay in the network had an effect on the data transmission component, but what exactly caused this situation?

As in the analysis of the RTT component in Chapter 6, I can use Maude's tracing facilities to examine this particular behaviour of the data transmission component from initial state `data2`. I include the appropriate tracing commands at the start of the file containing the module `NORM-DT-2`, and perform the above rewrite again, this time redirecting Maude's output to a new file. Analysing the trace of the rewrite reveals the events that lead to the above state:

- At time 1404, the router throws away the `DATA` message containing segment number 3 :: 1 because its buffer is full.
- `rec1` and `rec2` discover the loss at time 1740 and 1744 respectively, and they both set their `NACKbackoffTimer`.
- When their timers expire, the receivers each send a `NACK` message for the missing segment—`rec1` at time 2056, and `rec2` at time 2082. However, both repair requests are lost because the router's buffer is full.
- The holdoff timer of `rec1` expires at time 2692. The last segment is has received from the sender is 4 :: 11.
- At time 2718, the holdoff timer of `rec2` expires. The last segment it has received is 4 :: 13.
- At time 2787 the sender has sent its last data segment and the first `FLUSH` message for data objects 3 and 4. The `FLUSH` is thrown away by the router.

- `rec1` gets the sender's second `FLUSH` message at time 3397, and sets its `NACKbackoffTimer` attribute to 293.
- `rec2` receives the sender's second `FLUSH` at time 3401. The random backoff function sets the value of the `NACKbackoffTimer` attribute to zero, with the effect that `rec2` immediately generates a repair request for segment 3 :: 1.
- At time 3424, the sender transmits its fourth and final `FLUSH` message, and clears the content of its flush buffer.
- When the repair request of `rec2` finally reaches the sender at time 3453, the sender cannot provide the requested repair.

There was actually just *one* case of *data* packet loss in this behaviour, but several control and feedback messages were lost. I have a couple of remarks to this series of events:

1. If either of the two first `NACK` messages had reached the sender, it would have been able to provide repair for the missing segment. Furthermore, if the sender's first `FLUSH` message for objects 3 and 4 had not been lost, there would probably have been sufficient time for a repair request to reach the sender before it emptied is buffer content[2].
2. When a repair request finally got through to the sender, it had already finished its transmission. The difference in time between the state where the sender sends its final `FLUSH` and empties its buffer, and the state where it receives a `NACK` message is just 29 ms. If the sender had kept its data for a while after transmitting the final `FLUSH`, it would have been able to provide the needed repair.

In the first remark, I speculate on what *could* have happened if the `NACK` and `FLUSH` messages had not been dropped by the router. They where dropped, however, and it is not the job of the NORM protocol to prevent network errors. In Section 7.1.1, I pointed out that the NORM draft specification mentions, but does not define, a mechanism that allows receivers to self-initiate the NACK process "when no messages are received from the sender" [7]. Depending on how we interpret this sentence, we can imagine a mechanism whereby a receiver can repeat a `NACK` after a certain time if it did not receive the requested repair, or generate a `NACK` if it has not received any messages at all from the sender for some time. Remark number 1 points to two situations where such a mechanism would have been useful.

The second remark can serve to illustrate my discussion in Section 7.1.3 of an aspect of the specification of the FLUSH process: the NORM specification says that when the sender has reached the end of enqueued data, its transmits a series of FLUSH messages in order to get the receivers to ask for repair before it flushes the data. It seems logical to me as a nonexpert that after the sender has transmitted the final FLUSH, it ought to wait for some time before emptying the buffer, giving the receivers more time to respond.

[2] However, the resulting `NACK` messages and repair segments might of course have been dropped by the network.

8.6 Concluding Remarks

It is hard to draw any conclusions on the basis of this incomplete analysis of the data and repair transmission component. On the one hand, a rewrite of the state `data1` showed a behaviour of the component where the four data objects where successfully transmitted to the receivers, and where the protocol managed to repair the packets that were lost in the network. On the other hand, increasing the sender's transmission rate produced a behaviour where the loss of a data packet was not detected by the sender because the repair requests from the receivers were dropped by the network, and where the sender had finished the transmission by the time a repair request finally got through to it. However, the partial analysis brings us back to the questions that arose during the modelling of the component: How are we to interpret what appears to be an incomplete description of a procedure, and how should we model it? I chose to leave out the mechanism by which the receivers can self-initiate the NACK procedure, because the specification did not provide sufficient information about how it works. I also chose to model the FLUSH process exactly as it is described in the NORM draft, because it would not be correct to "add my own ideas" to what is in fact a definition of a procedure.

Chapter 9

Outlining the Combined Specification

In Chapters 5 and 7, I have specified the RTT and data transmission components of the NORM protocol as stand-alone Real-Time Maude specifications. In this chapter, I will describe how these specifications can be combined so that they can be executed simultaneously. Formalising the different functionalities of a distributed system as separate specifications is a good way of dealing with the complexity of such systems. When specifying a component, one only has to consider the actions that take place in that component, and in addition possible input from and output to other components. This makes it easier to produce a specification that mirrors the original design of the system correctly, and the specification will become more readable and easier to analyse.

The timed object-oriented module `NORM-COMBINED`, which can be found in Appendix B, combines the RTT and the data transmission specifications by declaring sender and receiver classes that inherit the rewrite rules that can be reused for combined execution of the specifications. The rewrite rules that were specified for stand-alone execution only, will have to be redefined. I will describe which parts of the specifications that will have to be redefined, but specifying the new rules is beyond the scope of this thesis. Finally, the module also contains equations that extend the operators `delta` and `mte` for the new sender and receiver classes.

9.1 Classes for the Combined Specification

The timed object-oriented module `NORM-COMBINED` imports the specifications for the RTT and data transmission components. Two new classes are declared in the module: `SenderCombined` and `ReceiverCombined`. The first is declared to be a subclass of `DTsender` and `RTTsender`, and the second a subclass of `DTreceiver` and `RTTreceiver`. Objects of these new classes inherit the attributes and rules defined for objects of their superclasses.

```
(tomod NORM-COMBINED is
  including NORM-RTT .
  including NORM-DATA-TRANSMISSION .
```

```
class SenderCombined .
subclass SenderCombined < DTsender .
subclass SenderCombined < RTTsender .


class ReceiverCombined .
subclass ReceiverCombined < DTreceiver .
subclass ReceiverCombined < RTTreceiver .
```

9.2 Redefining Rewrite Rules for the Combined Specification

In Section 4.2.5, I explained that if an event in the protocol leads to changes in both the RTT and the data transmission component, I will have to specify one set of rewrite rules for stand-alone execution of each subprotocol, and one set of rewrite rules for executing both components simultaneously. Events that only affect one component are specified as rules for the `RTTsender`, `DTsender`, `RTTreceiver`, and `DTreceiver` classes, and are inherited by objects of the `SenderCombined` and `ReceiverCombined` classes.

In the two Real-Time Maude specifications I have presented it is above all the rules involving the `NACK` message that have to be redefined in order to execute both simultaneously. The receivers use `NACK` messages primarily to ask for repair of one or more data segments, but if they are about to provide RTT feedback to the sender they will also include this information in the `NACK` message. The two rules below, the first from the data transmission specification and the second from the RTT specification, will have to be replaced by a single rule for objects of the class `ReceiverCombined`:

```
crl [makeNACKdecision] :
  < R : DTreceiverAlone |
    parent : O, NormRobustFactor : NZN, GRTT : T,
    CLR : B, sendersCurrTransPos : DUI,
    repairRequests : DUIL, extRepRequests : DUIL',
    NACKbackoffTimer : 0, NACKcycleHoldoffTimer : INF >
  =>
  (if not(alreadyNACKed(DUIL, DUIL')) and
     (DUI == 0 :: 0 or-else smallerThan(first(DUIL), DUI))
   then
   (< R : DTreceiverAlone |
      repairRequests : nil, extRepRequests : nil,
      NACKbackoffTimer : INF, NACKcycleHoldoffTimer : (NZN + 2) * T >
   (intoLink NACK(sublistUpToDUI(DUIL, DUI), 0, B) from R to O))
   else
   < R : DTreceiverAlone |
     repairRequests : nil, extRepRequests : nil,
     NACKbackoffTimer : INF > fi)
  if DUIL =/= nil .


rl [cancelACK1] :
  < R : RTTreceiverAlone |
    parent : O, clock : T, timestamp : T', receivedTimestamp : T'',
    ACKtimer : TI, ACKholdoffTimer : TI',
    NormRobustFactor : NZN, GRTT : N,
    nacksToBeSent : dly(NACK, 0) ++ ML, CLR : B >
  =>
  < R : RTTreceiverAlone |
    ACKtimer : (if TI =/= INF and TI =/= 0 then INF else TI fi),
    ACKholdoffTimer : (if TI =/= INF and TI =/= 0
                       then NZN * N else INF fi),
    nacksToBeSent : ML >
  (intoLink NACK(nil, (if T' == 0 then 0 else T' + (T monus T'') fi), B)
  from R to O) .
```

In the [makeNACKdecision] rule, the receiver checks whether it has any repair needs when its backoff timer expires, and sends a NACK message if it does. In the [cancelACK1] rule , the receiver cancels a pending RTT feedback message if it is about to send a NACK. In the new rule, the receiver will have to check whether it should also include RTT feedback every time it sends a repair request. Likewise, the rules where a sender receives a NACK message will have to be replaced by new rules for the combined protocol so that the sender can receive both repair requests and RTT feedback at the same time.

In addition to the rules treating the NACK message, I will have to define a new rule for data transmission to make sure that the sender has started collecting RTT information from the receivers *before* it transmits any data messages. The informal NORM protocol specification says that the sender can either start sending data segments immediately after sending a RTT feedback request, or it can wait until it has received a message from some receiver, to make sure that there are receivers present. When defining the rule, I will have to chose one of these options.

9.3 Extending the delta and mte Operators

Finally, the timed object-oriented module NORM-COMBINED contains equations that extend the operators delta and mte for the combined protocol. The equations for delta model the effect of time on objects of the classes SenderCombined and ReceiverCombined, and the equations for mte give the maximum possible time increase for objects of these classes.

```
   vars S R : Oid .
   vars T T' : Time .
   vars TI TI' TI'' TI''' TI'''' TI''''' : TimeInf .
```

```
  eq delta(< S : SenderCombined | clock : T, dataTransTimer : TI,
                  NACKaccumTimer : TI', repairCycleHoldoffTimer : TI'',
                  FLUSHtimer : TI''', SQUELCHholdoffTimer : TI'''',
                  CCtransTimer : TI''''' >, T') =
    < S : SenderCombined | clock : T + T', dataTransTimer : TI monus T',
            NACKaccumTimer : TI' monus T',
            repairCycleHoldoffTimer : TI'' monus T',
            FLUSHtimer : TI''' monus T',
            SQUELCHholdoffTimer : TI'''' monus T',
            CCtransTimer : TI''''' monus T' > .

  eq delta(< R : ReceiverCombined | clock : T, NACKbackoffTimer : TI,
                  NACKcycleHoldoffTimer : TI', ACKtimer : TI'',
                  ACKholdoffTimer : TI''' >, T') =
    < R : ReceiverCombined | clock : T + T', NACKbackoffTimer : TI monus T',
            NACKcycleHoldoffTimer : TI' monus T', ACKtimer : TI'' monus T',
            ACKholdoffTimer : TI''' monus T' > .


  eq mte(< S : SenderCombined | dataTransTimer : TI, NACKaccumTimer : TI',
                  repairCycleHoldoffTimer : TI'', FLUSHtimer : TI''',
                  SQUELCHholdoffTimer : TI'''', CCtransTimer : TI''''' >) =
    min(min(min(min(min(TI, TI'), TI''), TI'''), TI''''), TI''''') .

  eq mte(< R : ReceiverCombined | NACKbackoffTimer : TI,
                  NACKcycleHoldoffTimer : TI', ACKtimer : TI'',
                  ACKholdoffTimer : TI''' >) =
    min(min(min(TI, TI'), TI''), TI''') .

endtom)
```

Chapter 10

Summary and Conclusions

In this thesis I have applied formal methods to model and analyse a communication protocol. The purpose of my work has been threefold: to present an analysis of a specific protocol, the NORM multicast protocol; to investigate the usefulness of formal methods as part of the design process of distributed systems; and to evaluate the applicability of the Real-Time Maude specification formalism and tool in analysing a communication protocol. In this chapter, I summarise and discuss the results of this work. The first section is a summary of the specification and analysis effort. On the basis of this effort, Section 10.2 discusses the usefulness of the techniques I have applied in the process of designing and specifying distributed systems such as the NORM protocol. Finally, in Section 10.3, I discuss my experience with the Real-Time Maude language and tool.

10.1 Summary of Specification and Analysis Effort

This section summarises the results of the modelling and analysis effort, and shows which techniques lead to which results. A large part of the effort has consisted in formalising the RTT and data transmission components of the NORM specification draft as Real-Time Maude specifications. In analysing the specifications, I have made use of increasingly stronger formal methods for automatical analysis:

- simulating *one* possible behaviour of a component by executing the specification with the Real-Time Maude timed rewrite command,
- using the timed search commands for searching *all* possible behaviours (relative to the selected time sampling strategy) from an initial state for desired and undesired states, and
- analysing *all* possible behaviours (again, relative to the selected time sampling strategy) from some initial state by checking whether each behaviour satisfies a given temporal logic property.

Such analysis of *all* behaviours of a component from an initial state is usually not considered a *verification* of the component, since we do not check the property for all possible initial states. If we find an error during the analysis, we have proved that the property does not hold. However, if we do not find any error, we *cannot* say that the property holds for all possible initial states.

10.1.1 The Results of Modelling and Analysing the RTT Component

In the RTT component, the sender calculates the group's greatest round-trip time (GRTT), based on the highest round-trip time (RTT) values in the receiver group. A receiver principally uses the `ACK` message to provide feedback, but can also include RTT information in the `NACK` repair request message. The sender announces its GRTT estimate to the receiver set in every message it sends.

For the analysis, I defined two different initial states, `rtt1` and `rtt2`. Specifying the RTT component and analysing the Real-Time Maude specification w.r.t. the initial states, lead to the following results:

1. Results from specifying the component:

   - The informal specification states that a receiver will cancel its pending `ACK` message if it receives an `ACK` from another member of the group whose reception rate is close to or less than the receiver's rate. This information is also included in the `NACK` message; however, it is not used by the receivers to suppress their RTT feedback. It became apparent during the modelling that it is logical that the protocol should take advantage of the `NACK` message for further suppression of RTT feedback, since receiver feedback suppression is important to make NORM scalable. The current version of the NORM specification draft has been changed to make use of the reception rate information in the `NACK` message for suppression of `ACK` feedback.

2. Results from executing the specification:

   - Rewriting the RTT component from the two initial states produced behaviours where the sender had calculated a GRTT estimate as specified, and each receiver had received the estimate. In one of the behaviours, the peak RTT value of the receiver group increased. Examining the trace of the rewrite showed that the peak RTT increased because two messages entered the same link. This is a normal behaviour for the protocol.

3. Results from searching:

   - Searching through the behaviours from a state where the peak RTT had increased revealed that once the sender had recorded this value, it was not adjusted even though the peak RTT value of the receiver group decreased. Since the sender is supposed to adjust the GRTT estimate to changing network conditions, this was clearly not a correct behaviour. The behaviour can be traced back to the informal specification, where the algorithm describing how the sender will update its peak RTT tracking value is not fully specificed. In the current version of the NORM protocol, the algorithm has been considerably changed.

4. Results from model checking:

    - Model checking the RTT component from the initial state which caused an erroneous behaviour confirmed the results from the search: the sender's recorded peak RTT did not decrease even though the receiver group's highest RTT value did. However, model checking the component also showed that in every possible behaviour from the initial state, the sender reaches a stable GRTT estimate (due to the static topology), and every receiver gets this estimate.

To summarise, modelling and analysing the RTT component showed that it calculates a GRTT estimate and announces this to the receivers as intended, but that an important part of the algorithm for updating the estimate is missing. Furthermore, the component does not exploit the information in the `NACK` message for suppressing receiver feedback.

10.1.2 The Results of Modelling and Analysing the Data and Repair Transmission Components

The goal of the data transmission component is for the sender to reliably transmit one or more data objects, segmented into data packets, to a set of receivers. The receivers use `NACK` messages to request retransmission of lost segments.

I defined two slightly different initial states, `data1` and `data2`, for analysing the component. The state `data2` was specified with a large transmission delay in the network. Specifying the data transmission component, and analysing the Real-Time Maude specification w.r.t. the states `data1` and `data2` lead to the following results:

1. Results from specifying the component:

    - The informal specification says that a receiver can only initiate the repair request procedure in conjunction with certain events caused by the environment. However, the specification also mentions, but does not define, a procedure whereby a receiver can *self-initiate* a repair request. Since there is not enough information about this second option, it could not be included in the Real-Time Maude specification of the component. I have not found any more information about this option in the current version of the informal specification.
    - When the sender receives a repair request, it sets a timer and starts aggregating `NACK` messages in order to make repair transmission more efficient. The duration of this timeout is defined in the informal specification. However, the building block document specifies a *longer* timeout period. Since the current version of the informal specification also uses this larger timeout value, I assume that this value is the intended duration of the timeout, and use it in the Real-Time Maude specification of the component.

- When a sender reaches the end of transmission of enqueued data, it sends a series of FLUSH messages to invite the receivers to request repair for missing segments before is deletes the data content. The informal specification defines the duration of the interval *between* transmission of FLUSH messages. Given the meaning of the FLUSH message, it seems logical that the sender ought to wait a certain amount of time before deleting the data content *after* the final FLUSH message. However, no such time interval is described in the informal specification, and as a consequence, in the Real-Time Maude specification, the data content is cleared immediately after the final FLUSH message. The current version of the informal specification does not seem to mention such a time interval either.

2. Results from executing the specification:

   - Executing the specification from the initial state data1 produced a behaviour where transmission was successfully completed. Tracing the execution showed that there had been several cases of packet loss which had been repaired.
   - Executing the component from the initial state data2, which I thought more likely to lead to an error state, since there is a much larger delay in the network than in the state data1, produced a behaviour where the sender had finished data transmission, but the receivers were still missing data segments. The sender had received a repair request, but was unable to retransmit the requested segment because it had already cleared its data content. Tracing the execution made it possible to examine step by step the events that lead to the error state. The results of this execution highlight the importance of clarifying the parts of the informal specification that appeared not to be fully specified, in particular the transmission of FLUSH messages.

3. Results from searching:

   - Due to the nondeterminism in the specification, which leads to a combinatorial explosion in the state space, attempts at searching for undesired states from the initial state data1 within large time intervals failed. However, with a smaller time bound, it was possible to use the search command to find that no data packets were lost within the time interval. The `find latest` command, which from an initial state searches for the behaviour where it takes the longest time for a match with the search pattern to occur, and returns the first state of that behaviour which matches the pattern, proved very useful for finding a suitable time bound for the subsequent model checking analysis.

4. Results from model checking:

   - Time bounded model checking of the specification w.r.t. the initial state data1 showed that for the first ten data packets sent, the receivers sooner or later receive every packet.

To summarise, modelling and analysing the data and repair transmission component did not disclose any errors as such in the subprotocol's design, but revealed that there are inconsistencies in the original specification, and that the description of some of its procedures is incomplete.

10.2 Formal Modelling and Analysis of a Protocol under Development

Developing a computer system is a very complex and difficult design task. The purpose of formal methods is to provide techniques and tools for specification and analysis that can be integrated in the development process in order to avoid design errors in the system. Errors that are detected at an early stage in the design process are relatively inexpensive to correct, whereas the cost of fixing a bug once the system is implemented can be very high. If the error is not discovered until the product is released, the manufacturer may end up having to withdraw the product from the market: in 1995, the Intel Corporation had to replace the defective Pentium processor, at a cost of $ 500 million [11]. For very large systems, it may not be possible to model and analyse the entire system, and a more pragmatic approach is to apply formal methods to *critical* parts of the system.

In this thesis, I have demonstrated, by applying a specific formal method to a protocol specification draft, how such techniques and tools can contribute to the understanding of a complex distributed system under development. The NORM protocol specification examplifies how such systems are most often specified: in an informal style, using plain text or an informal modelling language such as UML. Such specifications usually contain crucial implicit assumptions, that may be self-evident to the designers. They will also often contain ambiguities and inconsistencies, and procedures may be underspecified. Formalising (parts of) the NORM multicast protocol as Real-Time Maude specifications disclosed several such ambiguities, inconsistencies, and cases of underspecification in the original specification. Formalising a system results in a *consistent* and *unambiguous* description of its functionalities, which can be subjected to formal analysis. Thus, one can analyse the consequences of design choices at an *early* stage in the development process, and use the results to improve the system's design before it is implemented.

Testing and analysing the Real-Time Maude specifications with the rewriting, search, and model checking tools proved to be very useful. Simulating the RTT component with the timed rewrite command produced a possible, and seemingly normal, behaviour where the RTT values changed due to increased traffic in a link. However, increased RTT values also revealed a flaw in the algorithm for updating the GRTT estimate, as became evident during further analysis of the component using the search and model checking tools. Having a range of increasingly stronger analysis tools is an advantage: rapid prototyping gives a "feel" for how the system functions, and with search and model checking tools one can actively look for possible erroneous behaviours in the specification.

When model checking a specification, one tries to verify that the specification w.r.t. an initial state satisfies a desired property, expressed as a logical formula. The desired properties that a system is supposed to satisfy are most often expressed informally in the specification. However, having a precise and unambiguous description of how the system *should* behave greatly simplifies the task of checking that it *does*. This is the function of a *requirements specification*, which specifies the desired properties of a system as e.g. temporal logical formulas. I have not found a formal requirements specification for the NORM protocol, but it would be a benefit for the understanding and analysis of the protocol.

10.3 Evaluating the Real-Time Maude Language and Tool

Since the purpose of formal methods is to develop formalisms and tools that are *useful* in the process of designing a system, the practical applicability of a formal method should be a criterion in evaluating it. Developers, and not just formal methods experts, should be able to learn and use the techniques during the design process. A formalism should be intuitive and easy to understand, and the analysis tools should be easy to use. After all, the focus should be on understanding and analysing the *problem*, not learning how to use the tool. This section evaluates the Real-Time Maude formalism and tool, based on my experiences from specifying the NORM multicast protocol.

The Real-Time Maude formalism is designed to allow a *natural* and *intuitive* representation of real-time systems. It is a general and flexible formalism, which makes it suitable for modelling many different kinds of systems and communication. It is also a simple language, which is easily learned. Unlike many other formalisms, Real-Time Maude lets the user specify both the static parts and the dynamic parts of a system in the *same* formalism, which greatly simplifies the modelling task. In addition, linear temporal logic requirements specifications can be written directly in the Real-Time Maude language and verified with the Real-Time Maude model checker, that is, both the *behavioural specification* of the system (its "model") and its *requirements specification* are written in the same language. The Maude formalism, and its extension Real-Time Maude, provides a natural way to model *concurrent* distributed systems, and the rewrite rule is a very intuitive representation of a (local) transition from one state of the system to another. Although the rewrite rules of my NORM specifications sometimes became quite large with many details (since there is a limit to how much you can abstract away from a transition), they were still easy to follow because of the intuitive form of the rewrite rule. In particular, Real-Time Maude supports a very intuitive *object-oriented* specification style, which is convenient for modelling distributed real-time systems.

Real-Time Maude gives the user complete freedom in specifying a suitable time domain when modelling a real-time system. It also provides some predefined time domains that can be imported into a specification. I chose to use a predefined discrete time domain because it was suitable for my modelling task. The case study of the AER/NCA protocol in [21] provided some useful general techniques for specifying real-time object-oriented systems, such as the `delta` and `mte` functions for modelling the effect of time, and calculating the maximal possible time increase in a configuration, respectively.

A great advantage with the Real-Time Maude system is that one can easily define different initial states for prototyping purposes, and for searching. This is a very flexible way of analysing a specification—it can be analysed for many different scenarios simply by defining new initial states. I found that the Real-Time Maude timed rewrite and search commands are very user-friendly. They have a simple syntax, yet are powerful and flexible commands. However, when using the Real-Time Maude search tools, there is no way of analysing the particular behaviour(s) which leads to the goal state. When performing a search in (core) Maude, it is possible to see the *path* of the rewrite which lead to the goal state by using a command called `show path`. This is a very useful and informative feature for analysis purposes. Unfortunately, Full Maude and its extension Real-Time Maude do not yet provide this feature, but I hope that future versions will include a `show path`-like command.

In Chapters 6 and 8, I used the Real-Time Maude trace facility several times in order to examine the rewrite path resulting from *rewriting* a specification. For long rewrites the result from tracing an execution can become extremely large and difficult to read, since the trace contains a lot of information about how the Maude interpreter applied the equations and rewrite rules. Although there are options that allow the user to exclude some of this information, the output still contains information that is not useful for a first analysis of the rewrite. It would be advantageous to have more flexible tracing facilities. For instance, the Real-Time Maude model checker generates a counterexample when it is not able to verify a property. This counterexample lists the timestamped states, together with the label of the rule applied in each transition, of a path which invalidated the property. This is sufficient information for an initial analysis of a behaviour, and should be a trace option.

Bibliography

[1] The Real-Time Maude web page: `http://www.ifi.uio.no/RealTimeMaude/`.

[2] The Petri Nets World web page: `http://www.daimi.au.dk/PetriNets/`.

[3] The web page of the IETF's reliable multicast transport group: `http://www.ietf.org/html.charters/rmt-charter.html`.

[4] The Maude web page: `http://maude.cs.uiuc.edu/`.

[5] B. Adamson, C. Bormann, M. Handley, and J. Macker. NACK-oriented reliable multicast protocol (NORM). Internet draft, IETF, January 2004. Available at `http://www.ietf.org/internet-drafts/draft-ietf-rmt-pi-norm-09.txt`.

[6] B. Adamson, C. Bormann, M. Handley, and J. Macker. NACK-oriented reliable multicast (NORM) building blocks. Internet draft, IETF, March 2003. Expired September 2003.

[7] B. Adamson, C. Bormann, M. Handley, and J. Macker. NACK-oriented reliable multicast protocol (NORM). Internet draft, IETF, March 2003. Expired September 2003.

[8] B. Adamson, C. Bormann, M. Handley, and J. Macker. NACK-oriented reliable multicast (NORM) building blocks. Internet draft, IETF, November 2003. Available at `http://www.ietf.org/internet-drafts/draft-ietf-rmt-bb-norm-08.txt`.

[9] R. Bruni and J. Meseguer. Generalized rewrite theories. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, editors, *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP 2003)*, volume 2719 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2003.

[10] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989. Preliminary version appeared as Digital Equipment Corporation Systems Research Center report no. 39, 1989.

[11] R. W. Butler, V. A. Carreño, B. L. Di Vito, K. J. Hayhurst, C. M. Holloway, P. S. Miner, G. Lüttgen, and C. Muñoz. NASA Langley's research and technology-transfer program in formal methods, 2000. Available at `http://shemesh.larc.nasa.gov/fm/`.

[12] E. M. Clarke and J. M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[13] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.

[14] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude 2.0 Manual, Version 1.0*, 2003. Available at `http://maude.cs.uiuc.edu/manual/`.

[15] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In F. Gadducci and U. Montanari, editors, *Fourth Workshop on Rewriting Logic and its Applications, WRLA '02*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002. Available at `http://maude.cs.uiuc.edu/papers/`.

[16] G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56:131–133, 1995.

[17] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *TACAS'96*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 1996.

[18] G. Lowe. Towards a completeness result for model checking of security protocols. *Journal of Computer Security*, 7(2-3):89–146, 1999.

[19] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[20] J. Meseguer. Membership algebra as a logical framework for equational specification. In *In 12th International Workshop on Recent Trends in Algebraic Development Techniques (WADT'97)*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer-Verlag, 1998.

[21] P. C. Ölveczky. *Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic*. PhD thesis, University of Bergen, 2000. Available at `http://maude.cs.uiuc.edu/papers/`.

[22] P. C. Ölveczky. *Real-Time Maude 2.0 Manual*, 2003. Available at `http://www.ifi.uio.no/RealTimeMaude/`.

[23] P. C. Ölveczky and S. Meldal. Specification and prototyping of network protocols in rewriting logic. In *Proc. NIK'98*, 1998.

[24] P. C. Ölveczky and J. Meseguer. Specifying and analyzing real-time object systems in Real-Time Maude. In P. Pettersson and S. Yovine, editors, *Proc. Workshop on Real-Time Tools, Aalborg University, Denmark, 2001*, 2001. Technical report 2001-14, Department of Information Technology, Uppsala University.

[25] P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285:359–405, 2002.

[26] P. C. Ölveczky and J. Meseguer. Real-Time Maude 2.0. In N. Martí-Oliet, editor, *Fifth International Workshop on Rewriting Logic and its Applications (Preliminary Version)*,

Electronic Notes in Theoretical Computer Science, pages 267–294. Elsevier, 2004. Available at `http://www.ifi.uio.no/RealTimeMaude`.

[27] P. C. Ölveczky and J. Meseguer. Specification and analysis of real-time systems using Real-Time Maude. In M. Wermelinger and T. Margaria-Steffen, editors, *Fundamental Approaches to Software Engineering (FASE)*, volume 2984 of *Lecture Notes in Computer Science*. Springer, 2004.

[28] L. L. Peterson and B. S. Davie. *Computer Networks - A Systems Approach.* Morgan Kaufmann Publishers, 2000.

[29] B. Quinn and K. Almeroth. IP multicast applications: Challenges and solutions. Request for Comments 3170, IETF, 2001.

[30] V. Roca. Un état de l'art sur les techniques de transmission multipoint fiable. Available at `http://www.inrialpes.fr/planete/people/roca/doc/jres01\_rm\_sota.html`, 2001.

[31] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.

Appendix A

The NORM Specification

Section A.1 of this appendix contains the original NORM multicast protocol specification. The specification was issued in March 2003, and expired in September 2003. Section A.2 contains the section from the NORM protocol building block document which specifies the GRTT measurement procedure. The building block document was also issued in March 2003, and expired in September 2003. Both the NORM protocol specification and the building block document have been replaced with newer versions, which can be found at the IETF web page at `http://www.ietf.org/html.charters/rmt-charter.html`.

I am very grateful to one of the authors of the NORM specification, Brian Adamson, for kindly having granted me the permission to include these documents in my thesis.

A.1 The NORM Protocol Specification

```
RMT Working Group                                       B. Adamson/NRL
INTERNET-DRAFT                                     C. Bormann/Tellique
draft-ietf-rmt-pi-norm-06                            M. Handley/ACIRI
Expires: September 2003                                  J. Macker/NRL
                                                            March 2003


            NACK-Oriented Reliable Multicast Protocol (NORM)

Status of this Memo

  This document is an Internet-Draft and is in full conformance with all
  provisions of Section 10 of RFC2026.

  Internet-Drafts are working documents of the Internet Engineering Task
  Force (IETF), its areas, and its working groups.  Note that other
  groups may also distribute working documents as Internet-Drafts.

  Internet-Drafts are draft documents valid for a maximum of six months
  and may be updated, replaced, or obsoleted by other documents at any
  time.  It is inappropriate to use Internet-Drafts as reference mate-
```

Abstract

This document describes the messages and procedures of the Negative-acknowledgement (NACK) Oriented Reliable Multicast (NORM) protocol. This protocol is designed to provide end-to-end reliable transport of bulk data objects or streams over generic IP multicast routing and forwarding services. NORM uses a selective, negative acknowledgement mechanism for transport reliability and offers additional protocol mechanisms to conduct reliable multicast sessions with limited "a priori" coordination among senders and receivers. A congestion control scheme is specified to allow the NORM protocol fairly share available network bandwidth with other transport protocols such as Transmission Control Protocol (TCP). It is capable of operating with both reciprocal multicast routing among senders and receivers and with asymmetric connectivity (possibly a unicast return path) from the senders to receivers. The protocol offers a number of features to allow different types of applications or possibly other higher level transport protocols to utilize its service in different ways. The protocol leverages the use of FEC-based repair and other IETF reliable multicast transport (RMT) building blocks in its design.

1.0 Introduction and Applicability

The Negative-acknowledgement (NACK) Oriented Reliable Multicast (NORM) protocol is designed to provide reliable transport of data from one or more sender(s) to a group of receivers over an IP multicast network. The primary design goals of NORM are to provide efficient, scalable, and robust bulk data (e.g. computer files, transmission of persistent data) transfer across possibly heterogeneous IP networks and topologies. The NORM protocol design provides support for distributed multicast session participation with minimal coordination among senders and receivers. NORM allows senders and receivers to dynamically join and leave multicast sessions at will with minimal overhead for control information and timing synchronization among participants. To accommodate this capability, NORM protocol message headers contain some common information allowing receivers to easily synchronize to senders throughout the lifetime of a reliable multicast session. NORM is designed to be self-adapting to a wide range of dynamic network conditions with little or no pre-configuration. The protocol is purposely designed to be tolerant of inaccurate timing estimations or lossy conditions that may occur many networks including mobile and wireless. The protocol is also designed to exhibit convergence and efficient operation even in situations of heavy packet loss and large queueing or transmission delays.

This document is a product of the IETF RMT WG and follows the guidelines provided in RFC 3269 [1]. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14, RFC 2119 [2].

1.1 NORM Delivery Service Model

A NORM protocol instance (NormSession) is defined within the context of participants communicating connectionless (e.g. Internet Protocol (IP) or User Datagram Protocol (UDP)) packets over a network using pre-determined addresses and host port numbers. Generally, the participants exchange packets using an IP multicast group address, but unicast transport may also be established or applied as an adjunct to multicast delivery. In the case of multicast, the participating NormNodes will communicate using a common IP multicast group address and port number that has been chosen via means outside the context of the given NormSession. Other IETF data format and protocol standards exist that may be applied to describe and convey the required "a priori" information for a specific NormSession (e.g. Session Description Protocol (SDP) [5], Session Announcement Protocol (SAP) [6], etc).

The NORM protocol design is principally driven with the assumption of a single sender transmitting bulk data content to a group of receivers. However, the protocol MAY operate with multiple senders within the context of a single NormSession. In initial implementations of this protocol, it is anticipated that multiple senders will transmit independently of one another and receivers will maintain state as necessary for each independent sender. However, in future versions of NORM, it is possible that some aspects of protocol operation (e.g. round-trip time collection) may provide for alternate modes allowing more efficient performance for applications requiring multiple senders.

NORM provides for three types of bulk data content objects (NormObjects) to be reliably transported. These types include:

1) static computer memory data content (NORM_OBJECT_DATA type),

2) computer storage files (NORM_OBJECT_FILE type), and

3) non-finite streams of continuous data content (NORM_OBJECT_STREAM type).

The distinction between NORM_OBJECT_DATA and NORM_OBJECT_FILE is simply to provide a "hint" to receivers in NormSessions serving multiple types of content as to what type of storage should be allocated for received content (i.e. memory or file storage). Other than that distinction, the two are identical, providing for reliable transport of finite (but potentially very large) units of content. These static data and file services are anticipated to be useful for multicast-based cache applications with the ability to reliably provide transmission of large quantities of static data. Other types of static data/file delivery services might make use of these transport object types, too. The use of the NORM_OBJECT_STREAM type is at the application's discretion and could be used to carry static data or file content also. The NORM reliable stream service opens up additional possibilities such as serialized reliable messaging or other unbounded, perhaps dynamically produced content. The NORM_OBJECT_STREAM provides for reliable transport analogous to that of the Transmission Control Protocol (TCP), although NORM receivers will be able to begin receiving stream content at any point in time. The applicability of this feature will depend upon the application.

The NORM protocol also allows for a small amount of "out-of-band" data (sent as NORM_INFO messages) to be attached to the data content objects transmitted by the sender. This readily-available "out-of-band" data allows multicast receivers to quickly and efficiently determine the nature of the corresponding data, file, or stream bulk content being transmitted. This allows application-level control of the receiver node's participation in the current transport activity. This also allows the protocol to be flexible with minimal pre-coordination among senders and receivers. The NORM_INFO content is designed to be atomic in that its size MUST fit into the payload portion of a single NORM message.

NORM does _not_ provide for global or application-level identification of data content within in its message headers. Note the NORM_INFO out-of-band data mechanism could be leveraged by the application for this purpose if desired, or identification could alternatively be embedded within the data content. NORM does identify transmitted content (NormObjects) with transport identifiers that are applicable only while the sender is transmitting and/or repairing the given object. These transport data content identifiers (NormTransportIds) are assigned in a monotonically increasing fashion by each NORM sender during the course of a NormSession. Each sender maintains its NormTransportId assignments independently so that individual NormObjects may be uniquely identified during transport with the concatenation of the sender session-unique identifier (NormNodeId) and the assigned NormTransportId. The NormTransportIds are assigned from a large, but fixed, numeric space in increasing order and may be reassigned during long-lived sessions. The NORM protocol provides mechanisms so that the sender application may terminate transmission of data content and inform the group of this in an efficient manner. Other similar protocol control mechanisms (e.g. session termination, receiver synchronization, etc) are specified so that reliable multicast application variants may construct different, complete bulk transfer communication models to meet their goals.

In summary, the NORM protocol's goal is to provide reliable transport of different types of data content (including potentially mixed types). The senders enqueue and transmit bulk content in the form of static data or files and/or non-finite, ongoing stream types. The sender will provide for repair transmission of this content in response to NACK messages received from the receiver group. Mechanisms for "out-of-band" information and other transport control mechanisms are specified for use by applications to form complete reliable multicast solutions for different purposes.

1.2 NORM Scalability

Group communication scalability requirements lead to adaptation of negative acknowledgement (NACK) based protocol schemes when feedback for reliability is required [7]. NORM is a protocol centered around the use of selective NACKs to request repairs of missing data. NORM provides for the use of packet-level forward error correction (FEC) techniques for efficient multicast repair and optional proactive transmission robustness[8]. FEC-based repair can be used to greatly reduce the quantity of reliable multicast repair requests and repair transmissions[9]. The principal factor in NORM scalability is the volume of feedback traffic generated by the receiver set to facilitate reliability and congestion control. NORM uses probabilistic suppression of redundant feedback based on exponentially distributed random backoff timers. The performance of this type of suppression relative to other techniques is described in [10]. NORM dynamically measures the group's roundtrip timing status to set its suppression and other protocol timers. This allows NORM to scale well while maintaining reliable data delivery transport with low latency relative to the network topology over which it is operating. Feedback messages can be either multicast to the group at large or sent via unicast routing to the sender. In the case of unicast feedback, the sender "advertises" the feedback state to the group to facilitate feedback suppression. In typical Internet environments, it is expected that the NORM protocol will readily scale to group sizes on the order of tens of thousands of receivers. A study of the quantity of feedback for this type of protocol is described in [11]. NORM is able to operate with a smaller amount of feedback than a single TCP connection, even with relatively large numbers of receivers. Thus, depending upon the network topology, it is possible that NORM may scale to larger group sizes. With respect to computer resource usage, the NORM protocol does _not require that state be kept on all receivers in the group. NORM senders maintain state only for receivers providing explicit congestion control feedback. NORM receivers must maintain state for for each active sender. This may constrain the number of simultaneous senders in some uses of NORM.

1.3 NORM Environmental Requirements and Considerations

All of the environmental requirements and considerations that apply to the RMT FEC Building Block and the the RMT TCP-Friendly Multicast Congestion Control (TFMCC) Building Block also apply to NORM. When the RMT GRA Building Block is used with NORM, its environmental requirements and considerations SHALL also apply.

The NORM protocol SHALL be capable of operating in an end-to-end fashion with no assistance from intermediate systems beyond basic IP multicast group management, routing, and forwarding services. The NORM protocol SHOULD be compatible with techniques like Generic Router Assist (GRA) [12] for performance benefits when applicable. While the techniques utilized in NORM are principally applicable to "flat" end-to-end IP multicast multicast topologies, they could also be applied in the sub-levels of hierarchical (e.g. tree-based) multicast distribution if so desired. NORM can make use of reciprocal (among senders and receivers) multicast communication under the Any-Source Multicast (ASM) model defined in RFC 1112 [13], but SHALL also be capable of scalable operation in asymmetric topologies such as Source Specific Multicast (SSM) [14] where there may only be unicast routing service from the receivers to the sender(s).

NORM is compatible with IPv4 and IPv6. Additionally, NORM may be used with networks employing Network Address Translation (NAT) providing the NAT device supports IP multicast and/or can cache UDP traffic source port numbers for remapping feedback traffic from receivers to the sender(s).

2.0 NORM Architecture Definition

A NormSession is comprised of participants (NormNodes) acting as senders and/or receivers. NORM senders transmit data content in the form of NormObjects to the session destination address and the NORM receivers attempt to reliably receive the transmitted content using negative acknowledgments to request repair. Each NormNode within a NormSession is assumed to have a preselected unique 32-bit identifier (NormNodeId). NormNodes MUST have uniquely assigned identifiers within a single NormSession to distinquish between possible multiple senders and to distinguish feedback information from different receivers. There are two reserved NormNodeId values. A value of 0x00000000 is considered an invalid NormNodeId value and a value of 0xffffffff is a "wildcard" NormNodeId. Whilte, the protocol does not preclude multiple sender nodes concurrently transmitting within the context of a single NORM session (i.e. many- to-many operation), any type of interactive coordination among NORM senders is assumed to be controlled by the application or higher protocol layer. There are some optional mechanisms specified in this document which can be leveraged for such application layer coordination.

As previusly noted, NORM allows for reliable transmission of three different basic types of data content. The first type is NORM_OBJECT_DATA which is used for static, persistent blocks of data content maintained in the sender's application memory storage. The second type is NORM_OBJECT_FILE which corresponds to data stored in the sender's non-volatile file system. The NORM_OBJECT_DATA and NORM_OBJECT_FILE types both represent "NormObjects" of finite but potentially very large size. The third type of data content is NORM_OBJECT_STREAM which corresponds to an ongoing transmission of undefined length. This is analogous to the reliable streaming content provide by TCP for unicast data transport. The format of the stream content is application-defined and may be byte or message oriented. The NORM protocol provides for "flushing" of the stream to expedite delivery or possible enforce application message boundaries. NORM protocol implementations may offer either (or both) in-order delivery of the stream data to the receive application or out-of-order (more immediate) delivery of received segments of the stream to the receiver application. In either case, NORM sender and receiver implementations provide buffering to facilitate repair of the stream as it is transported. All NormObjects are logically segmented into FEC coding blocks and segments for transmission by the sender.

NormObjects and associated transmission segments are temporarily yet uniquely identified within the NormSession context using the given sender's NormNodeId and a temporarily unique NormObjectTransportId. These data content identifiers are sender-assigned and applicable and valid only during a NormObject's actual _transport_ (i.e. for as long as the sender is transmitting and providing repair of the indicated

NormObject). For a long-lived session, the NormObjectTransportId field can wrap and previously-used identifiers may be re-used. Note that globally unique identification of transported data content is not provided by NORM and, if required, must be managed by the NORM application. Individual NormObject segments are further identified with FEC coding block and symbol (segment) indentifiers. This is discussed in detail later in this document.

2.1 NORM Protocol Operation Overview

A NORM sender primarily generates messages of type NORM_DATA that carry the NormObject data content segments and related FEC parity-based repair segments for the bulk data/file or stream objects being transferred. By default, FEC segments are sent only in response to receiver repair requests (NACKs) and thus normally impose no additional transmission overhead. However, the NORM implementation MAY be optionally configured to proactively transmit some amount of FEC segments along with the data content to potentially enhance performance (e.g., improved delay) at the cost of additional overhead with initial data transmission. This configuration may be sensible for certain network conditions and can allow for robust, asymmetric multicast (e.g., unidirectional routing, satellite, cable) [19] with reduced receiver feedback, or, in some cases, no feedback.

A sender message of type NORM_INFO is also defined and is used to carry any optional "out-of-band" context information for a given transport object. A single NORM_INFO message can be associated with a NormObject. Because of its atomic nature, missing NORM_INFO messages can be NACKed and repaired with a slightly lower delay process than NORM's general FEC-encoded data content. NORM_INFO may serve special purposes for some bulk transfer, reliable multicast applications where receivers join the group mid-stream and need to ascertain contextual information on the current content being transmitted. The NACK process for NORM_INFO will be described later.

The sender also generates messages of type NORM_CMD to assist in certain protocol operations such as congestion control, end-of-transmission flushing, round trip time estimation, receiver synchronization, and optional positive acknowledgement requests or application defined commands. The transmission of NORM_CMD messages from the sender is accomplished by one of three different processes. These are: single, best effort unreliable transmission of the command; repeated redundant transmissions of the command; and positively-acknowledged commands. The transmission technique used for a given command depends upon the function of the command. Several core commands are defined for basic protocol operation. Additionally, implementations MAY wish to consider providing the OPTIONAL application-defined commands that can take advantage of the transmission methodologies available for commands. This allows for application-level session management mechanisms which can make use of information available to the underlying NORM protocol engine (e.g. round-trip timing, transmission rate, etc).

NORM receivers generate messages of type NORM_NACK or NORM_ACK in response to transmissions of data and commands from a sender. The NORM_NACK messages are generated to request repair of detected data transmission losses. Receivers generally detect losses by tracking the sequence of transmission from a sender. Sequencing information is embedded in the transmitted data packets and end-of-transmission commands from the sender. NORM_ACK messages are generated in response to certain commands transmitted by the sender. In the general (and most scalable) protocol mode, NORM_ACK messages are sent only in response to congestion control commands from the sender. The feedback volume of these congestion control NORM_ACK messages is controlled using the same timer-based probabilistic suppression techniques as for NORM_NACK messages to avoid feedback implosion. In order to meet potential application requirements for positive acknowledgement from receivers, other NORM_ACK messages are defined and available for use.

All sender and receiver transmissions are subject to rate control governed by a peak transmission rate set for each participant by the application. This can be used to limit the quantity of multicast data transmitted by the group. When NORM's congestion control algorithm is enabled the rate for senders is automatically adjusted. In some networks, it may be desirable to establish minimum and maximum bounds for the rate adjustment depending upon the application even when dynamic congestion control is enabled. However, in the case of the general Internet, congestion control policy SHALL be observed which is compatible with coexistent TCP flows.

2.2 NORM Protocol Building Blocks

The operation of the NORM protocol is based upon the concepts presented in the Nack-Oriented Reliable Multicast (NORM) Building Block document[15]. This includes the basic NORM architecture and the data transmission, repair, and feedback strategies discussed in that document. NORM also makes use of Forward Error Correction encoding techiques for repair messaging and optional transmission robustness as described in [16]. NORM uses the FEC Payload ID as specified by the FEC Building Block Document[17]. Additionally, for congestion control, the NORM protocol specifies a mechanism based on the TCP-Friendly Multicast Congestion Control (TFMCC) Building Block described in [18].

2.3 NORM Design Tradeoffs

While the various features of NORM are designed to provide some measure of general purpose utility, it is important to emphasize the understanding that "no one size fits all" in the reliable multicast transport arena. There are numerous engineering tradeoffs involved in reliable multicast transport design and this requires an increased awareness of application and network architecture considerations. Performance requirements affecting design can include: group size, heterogeneity (e.g., capacity and/or delay), asymmetric delivery, data ordering, delivery delay, group dynamics, mobility, congestion control, and transport across low capacity connections. NORM contains various parameters to accommodate many of these differing requirements. The NORM protocol and its mechanisms MAY be applied in multicast applications outside of bulk data transfer, but there is an assumed model of bulk transfer transport service that drives the trade-offs that determine the scalability and performance described in this document.

The ability of NORM to provide reliable data delivery is also governed by any buffer constraints of the sender and receiver applications. NORM protocol implementations SHOULD be designed to operate with the greatest efficiency and robustness possible within application-defined buffer constraints. Buffer requirements for reliability, as always, are a function of the delay-bandwidth product of the network topology. NORM performs best with additional buffering as compared to typical point-to-point transport NORM feedback suppression based upon randomly-delayed transmissions from the receiver set. There are definitive tradeoffs between buffer utilization, group size scalability, and efficiency of performance. Large buffer sizes allow the NORM protocol to perform most efficiently in large delay-bandwidth topologies and allow for longer feedback suppression backoff timeouts. This yields improved group size scalability. NORM can operate with reduced buffering but at a cost of decreased efficiency (lower relative goodput) and reduced group size scalability.

3.0 Conformance Statement

This Protocol Instantiation document, in conjunction with the following Building Block documents identified in [15], [16], [17], and [18] completely specifies a working reliable multicast transport protocol that conforms to the requirements described in RFC 2357 [3].

4.0 NORM Message Formats

As mentioned in Section 2.1, there are two primary classes of NORM messages: sender messages and receiver messages. NORM_CMD, NORM_INFO, and NORM_DATA message types are generated by senders of data content, and NORM_NACK and NORM_ACK messages generated by receivers within a NormSession. An auxillary message type of NORM_REPORT is also provided for experimental purposes. This section described the message formats used by the NORM protocol. These messages and their fields are referenced in the detailed functional description of the NORM protocol given in Section 5.0. Individual NORM messages are designed to be compatible with the MTU limitations of encapsulating Internet protocols including IPv4, IPv6, and UDP. The current NORM protocol specification assumes UDP encapsulation and leverages the transport features of UDP. The NORM messages are independent of network addresses and can be used in IPv4 and IPv6 networks.

4.1 NORM Common Message Header

There are some common message fields contained in all NORM message types. All NORM protocol messages begin with a common header with information fields as follows:

NORM Common Message Header Format:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    version    |     type      |           sequence            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           source_id                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              ...                              |
```

The "version" field is a 8-bit value indicating the protocol version number. Currently, NORM implementations SHOULD ignore received messages with a different protocol version number than their own. This number is intended to indicate and distinguish upgrades of the protocol which may be non-interoperable.

The message "type" field is a 8-bit value indicating the NORM protocol message type. These types are defined as follows:

Message	Value
NORM_INFO	1
NORM_DATA	2
NORM_CMD	3
NORM_NACK	4
NORM_ACK	5
NORM_REPORT	6

The "sequence" field is a 16-bit value that is set by the message originator as a monotonically increasing number incremented with each NORM message transmitted to the session's destination address. The "sequence" field SHOULD not be incremented for messages not sent to the session group address (e.g. unicast NACKs or unicast ACKs). This value can be monitored by receiving nodes to detect packet losses in the transmission from a sender. Note that this value is NOT used in the NORM protocol to detect missing reliable data content and does NOT identify the application data or FEC payload that may be attached. This sequence number is intended for use in estimating raw packet loss for congestion control purposes. The size of this field is intended to be sufficient to allow detection of a reasonable range of packet loss within the delay-bandwidth product of expected network connections.

The "source_id" field is a 32-bit value identifying the node that sent the message. A participant's NORM node identifier (NormNodeId) can be set according to the application needs but unique identifiers must be assigned within a single NormSession. In some cases, use of the host IP address or a hash of it can suffice, but alternative methodologies for assignment and potential collision resolution of node identifiers within a multicast session need to be considered. For example, the "source identifier" mechanism defined in the Real-Time Protocol (RTP) specification [20] may be applicable to use for NORM node identifiers. At this point in time, the protocol makes no assumptions about how these unique identifiers are actually assigned.

4.2 NORM Sender Messages

NORM sender messages include the NORM_DATA type, the NORM_INFO type, and the NORM_CMD type. NORM_DATA and NORM_INFO messages contain application data content while NORM_CMD messages for various protocol functions.

4.2.1 NORM_DATA Message

The NORM_DATA message is expected to be the predominant type transmitted by NORM senders. These messages are used to encapsulate segmented data content for objects of type NORM_OBJECT_DATA, NORM_OBJECT_FILE, and NORM_OBJECT_STREAM. NORM_DATA messages may contain original or FEC-encoded application data content. The payload size of these messages SHALL be limited to a maximum of the sender's NormSegmentSize. A sender's NormSegmentSize is assumed to be constant for the duration of a given sender's term of participation in the session. The NormSegmentSize is expected to be configurable by the sender application prior to session participation as needed for network topology maximum transmission unit (MTU) considerations. For IPv6, MTU discovery may be leveraged at session startup

NORM_DATA Message Format

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    version    |  type = 2     |           sequence            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           source_id                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     flags     |     grtt      |     gsize     | fec_id = 129  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       fec_instance_id         |        fec_num_parity         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      fec_max_block_len        |         segment_size          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     object_transport_id       |      object_size (msb)        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       object_size (lsb)                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       fec_block_number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         fec_block_len         |         fec_symbol_id         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          payload_len*         |          offset (msb)*        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         offset (lsb)*                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         payload_data*                         |
```

*Note: The "payload_len" and "offset" fields for NORM_DATA messages containing parity information are actually values computed from FEC encoding of the "payload_len" and "offset" fields of the data segments of the applicable coding block. So, for parity segments, these do _not_ represent actual values. Parity packets can be identified as packets where "fec_symbol_id >= fec_block_len".

The "version", "type", "sequence", and "source_id" fields form the NORM Common Message Header asdescribed in Section 4.1.

The "flags" field contains a number of different binary flags providing information and hints regarding how the receiver should handle the identified object. Defined flags in this field include:

Flag	Value	Purpose
NORM_FLAG_REPAIR	0x01	Indicates message is a repair transmission
NORM_FLAG_EXPLICIT	0x02	Indicates a repair segment intended which meets a specific receiver erasure, as compared to parity segments provided by the sender for general purpose (with respect to an FEC coding block) erasure filling.
NORM_FLAG_INFO	0x04	Indicates availability of NORM_INFO for object
NORM_FLAG_UNRELIABLE	0x08	Indicates that repair transmissions for the specified object will be unavailable. (One-shot, best effort transmission)
NORM_FLAG_FILE	0x10	Indicates object is "file-based" data (hint to use disk storage for reception)
NORM_FLAG_STREAM	0x20	Indicates object is of type NORM_OBJECT_STREAM.

The NORM_FLAG_REPAIR flag is set when the associated message is a repair transmission. This information can be used by receivers to help observe a join policy where it is desired that newly joining receivers only begin participating in the NACK process upon receipt of new (non-repair) data content. The NORM_FLAG_EXPLICIT flag is used to mark repair messages sent when the data sender has exhausted its ability to provide "fresh" (previously untransmitted) parity segments as repair. This flag may be used by intermediate systems implementing Generic Router Assist (GRA) functionality to control subcasting of repair content to different legs of a reliable multicast topology with disparate repair needs. The NORM_FLAG_INFO flag is set only when there optional NORM_INFO content is available for the associated object. Thus, receivers will NACK for retransmission of NORM_INFO only when it is available. The NORM_FLAG_UNRELIABLE flag is set when the sender wishes to transmit an object with only "best effort" delivery and will not supply repair transmissions for the object. The NORM_FLAG_FILE flag can be set as a "hint" from the sender that the associated object should be stored in non-volatile storage. The NORM_FLAG_STREAM flag is set when the identified object is of type NORM_OBJECT_STREAM.

The "grtt" field contains a non-linear quantized representation of the sender's current estimate of group round-trip time (GRTT) (This is also referred to as R_max in the TFMCC Building Block [18]). This value is used to control timing of the NACK repair process and other aspects of protocol operation as described in this document. The algorithm for encoding and decoding this field is described in the RMT NORM Building Block document[15].

The "gsize" field contains a representation of the sender's current estimate of group size. This value is used to control feedback suppression mechanisms within the protocol for more optimized performance for different group sizes. The 8-bit "gsize" field consists of 4 bits of mantissa in the 4 most significant bits and 4 bits of base 10 exponent (order of magnitude) information in the 4 least significant bits. For example, to represent an approximate group size of 100 (or 1e02), the value of the upper 4 bits is 0x01 (to represent the mantissa of 1) and the lower 4 bits value would be 0x02 for an 8-bit representation of "0x12". As another example, a group size of 9000 (9e03) would be represented by the value 0x93. The group size does not need to be represented with a high degree of precision to appropriately scale backoff timers, etc.

The "fec_id" field corresponds to the FEC Encoding Identifier described in the FEC Building Block document [17]. Note the packet format illustrated above assumes "Small Block Systematic Codes" that corresponds to an FEC Encoding Identifier equal to 129. The other "fec_" fields may be interpreted or sized differently to supportother FEC Encoding Identifier types in the future.

The "fec_instance_id" corresponds to the "FEC Instance ID" of the FEC Object Transmission Informatiom given in the FEC Building Block document[17]. The "fec_instance_id" SHALL be a value corresponding to the particular type of Small Block Systematic Code being used (e.g. Reed-Solomon GF(2^8), Reed-Solomon GF(2^16), etc). The standardized assignment of FEC Instance ID values is described in [17].

The "fec_num_parity" corresponds to the "maximum number of of encoding symbols that can be generated for any source block" as described in for FEC Object Transmission Information for Small Block Systematic Codes in the FEC Building Block document [17]. For example, Reed-Solomon codes may be arbitrarily shortened to create different code variations for a given block length. In the case of Reed-Solomon (GF(2^8) and GF(2^16) codes, this value indicates the maximum number of parity segments available from the sender for the coding blocks. This field MAY be interpreted differently for other systematic codes as they are defined.

The "fec_max_block_len" indicates the current maximum number of user data segments per FEC coding block to be used by the sender during the session. This allows receivers to allocate appropriate buffer space for buffering blocks transmitted by the sender.

The "segment_size" field indicates the sender's current setting for maximum message payload content (in bytes). This allows receivers to allocate appropriate buffering resources and to determine other information in order to properly process received data messaging.

The "object_transport_id" field is a monotonically and incrementally increasing value assigned by a sender to the object being transmitted. Transmissions and repair requests related to that object use the same "object_transport_id" value. For sessions of very long or indefinite duration, the "object_transport_id" field may be repeated, but it is presumed that the 16-bit field size provides an adequate enough sequence space to prevent temporary object confusion amongst receivers and sources (i.e. receivers SHOULD re-synchronize with a server when receiving object sequence identifiers sufficiently out-of-range with the current state kept for a given source). During the course of its transmission within a NORM session, an object is uniquely identified by the concatenation of the sender "node_id" and the given "object_transport_id". Note that NORM_INFO messages associated with the identified object carry the same "object_transport_id" value.

The 48-bit "object_size" field indicates the total size of the object (in bytes) for the static object types of NORM_OBJECT_FILE and NORM_OBJECT_DATA. This information is used by receivers to determine storage requirements and/or allocate storage for the received object. Receivers with insufficient storage capability may wish to forego reliable reception (i.e. not NACK for) of the indicated object. In the case of objects of type NORM_OBJECT_STREAM, the "object_size" field is used to by the sender to indicate the size of its stream buffer to the receiver group. In turn, the receivers SHOULD use this information to allocate a stream buffer for reception of corresponding size.

The "fec_block_number", "fec_block_len", and "fec_symbol_id" fields correspond to the "Source Block Number", "Source Block Length, and "Encoding Symbol ID" fields of the FEC Payload ID format given by the FEC Building Block document[17]. The "fec_block_number" identifies the coding block's relative position with a NormObject. Note that, for NormObjects of type NORM_OBJECT_STREAM, the "fec_block_number" may wrap for very long lived sessions. The "fec_block_len" indicates the number of user data segments in the identified coding block. Given the "fec_block_len" (Source block length) information of how many symbols of application data is contained in the block, the receiver can determine whether the attached segment is data or parity content and treat it appropriately. The "fec_symbol_id" identifies which specific symbol (segment) within the coding block the attached payload conveys. Depending upon the value of the "fec_symbol_id" and the associated "fec_block_len" and "fec_num_parity" parameters for the block, the symbol (segment) referenced may be a user data or an FEC parity segment. For systematic codes, symbols numbered 0 through (fec_block_len-1) contain application data while segments numbered (fec_block_len) through (fec_block_len+fec_num_parity-1) contain the parity symbols calculated for the block.

The concatenation of object_tranport_id::fec_block_number::fec_symbol_id can be viewed as a unique transport data unit (TPDU) identifier for the attached segment with respect to the NORM sender.

The "payload_len" and "offset" fields are used to identify the relative position and quantity of the content of the message payload. For senders employing systematic FEC encoding, these fields will correspond to actual length and offset values for NORM_DATA messages which contain original data content. For NORM_DATA messages containing calculated parity content, these fields will actually contain values computed by FEC encoding of the "payload_len" and "offset" values of the NORM_DATA segments of the corresponding FEC coding block. Thus, the "payload_len" and "offset" values of missing data content can be determined when decoding an FEC coding block.

The "payload_data" field contains original data or computed parity content of the identified segment. The maximum length of this field corresponds to the sender's NormSegmentSize. The length of this field for messages containing parity content will always be of the length NormSegmentSize. When encoding data segments of varying sizes, the FEC encoder SHALL assume zero value padding for data segments with length less than the NormSegmentSize. The receiver will use the "payload_len" information to properly retrieve received data content and deliver it to the application.

4.2.2 NORM_INFO Message

The NORM_INFO message is used to convey OPTIONAL, application-defined, "out-of-band" context information for transmitted NormObjects. An example NORM_INFO use for bulk file transfer is to place MIME type information for the associated file, data, or stream object into the NORM_INFO payload. Receivers may use the NORM_INFO content to make a decision as whether to participate in reliable reception of the associated object. Each NormObject can have an independent unit of NORM_INFO associated with it. NORM_DATA messages contain a flag to indicate the availability of NORM_INFO for a given NormObject. NORM receivers may NACK for retransmission of NORM_INFO when they have not received it for a given NormObject. The size of the NORM_INFO content is limited to that of a single NormSegmentSize for the given sender. This atomic nature allows the NORM_INFO to be rapidly and efficiently repaired within the NORM reliable transmission process.

When NORM_INFO content is available for a NormObject, the NORM_FLAG_INFO flag SHALL be set in NORM_DATA messages for the corresponding "object_transport_id" and the NORM_INFO message shall be transmitted as the first message for the NormObject.

NORM_INFO Message Format

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    version    |  type = 1     |           sequence            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           source_id                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     flags     |     grtt      |     gsize     | fec_id = 129  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       fec_encoding_name       |        fec_num_parity         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       fec_max_block_len       |         segment_size          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      object_transport_id      |       object_size (msb)       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       object_size (lsb)                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         payload_data                          |
```

The "version", "type", "sequence", and "source_id" fields form the NORM Common Message Header asdescribed in Section 4.1.

The "flags", "grtt", "gsize", "fec_id", "fec_encoding_name", "fec_num_parity", "fec_max_block_len", "segment_size", "object_transport_id", and "object_size" fields carry the same information and serve the same purpose as with NORM_DATA messages. These values allow the receiver to prepare appropriate buffering, etc, for further transmissions from the sender when NORM_INFO is the first message received.

The NORM_INFO "payload_data" field contains sender application-defined content which can be used by receiver applications for various purposes as described above.

4.2.3 NORM_CMD Message

NORM_CMD messages are transmitted by senders to perform a number of different protocol functions. This includes functions such as round-trip timing collection, congestion control functions, synchronization of sender/receiver repair "windows", and notification of sender status. A core set of NORM_CMD messages is enumerated. Additionally, a range of command types remain available for potential application-specific use. Some NORM_CMD types may have dynamic content attached. Any attached content will be limited to maximum length of the sender NormSegmentSize to retain the atomic nature of commands. All NORM_CMD message begins with a common set of fields, after the usual NORM message common header. The standard NORM_CMD fields are:

NORM_CMD Standard Fields

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    version    |   type = 3    |           sequence            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           source_id                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      grtt     |     gsize     |    flavor     |          ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The "version", "type", "sequence", and "source_id" fields form the NORM Common Message Header as described in Section 4.1.

The "grtt" and "gsize" fields provide the same information and serve the same purpose as with NORM_DATA and NORM_INFO messages. The "flavor" field indicates the type of command to follow. The remainder of the NORM_CMD message is dependent upon the command type ("flavor"). The command flavors include:

Command	Flavor Value	Purpose
NORM_CMD(FLUSH)	1	Used to indicate sender temporary or permanent end-of-transmission. (Assists in robustly initiating outstanding repair requests from receivers).

NORM_CMD(SQUELCH)	2	Used to advertise sender's current repair window in response to out-of-range NACKs from receivers.
NORM_CMD(ACK_REQ)	3	Used to request positive acknowledgement from a list of receivers.
NORM_CMD(REPAIR_ADV)	4	USed to advertise sender's aggregated repair state for suppression of unicast receiver feedback.
NORM_CMD(CC)	5	Used for GRTT measurement and explicitly collection of congestion control feedback.
NORM_CMD(APPLICATION)	6	Used for application-defined purposes which may need to temporarily preempt data transmission.

NORM_CMD(FLUSH) Message

The NORM_CMD(FLUSH) command is sent when the sender reaches the end of all data content and pending repairs it has queued for transmission. This command is repeated once per 2*GRTT to excite the receiver set for any outstanding repair requests up to and including the transmission point indicated within the NORM_CMD(FLUSH) message. The number of repeats is equal to NORM_ROBUST_FACTOR. The greater the NORM_ROBUST_FACTOR, the greater the probability that all applicable receivers will be excited for repair requests (NACKs) _and_ that the corresponding NACKs are delivered to the sender. If a NORM_NACK message interrupts its flush process, the sender will re-initiate the flush process when any resulting repair transmissions are completed. Note that receivers also employ a timeout mechanism to self-initiate NACKing when no messages are received from a sender. This inactivity timeout is related to 2*GRTT*NORM_ROBUST_FACTOR and will be discussed more later. With a sufficient NORM_ROBUST_FACTOR value, data content is delivered with a high assurance of reliability. The penalty of a large NORM_ROBUST_FACTOR value is potentially excess sender NORM_CMD(FLUSH) transmissions and a longer timeout for receivers to self-initiate the terminal NACK process.

For finite-size transport objects such NORM_OBJECT_DATA and NORM_OBJECT_FILE, the flush process (if there are no further pending transmissions) will occur at the end of these objects and thus any FEC repair information is available for repairs in response to repair requests elicited by the flush command. However, for NORM_OBJECT_STREAM, the flush may occur at any time, including in the middle of an FEC coding block if systematic FEC codes are emplyed. In this case, the sender will not yet be able to provide FEC parity content as repair for the concurrent coding block and will be limited to explicitly repairing stream data content for that block.

Applications that anticipate frequent flushing of stream content SHOULD be judicious in the selection of the FEC coding block size (i.e. do not use a very large coding block size if frequent flushing occurs). For example, a reliable multicast application transmitting an on-going series of intermittent, relatively small messaging content will need to trade-off using the NORM_OBJECT_DATA paradigm versus the NORM_OBJECT_STREAM paradigm with an appropriate FEC coding block size. This is analogous to application trade-offs for other transport protocols such as the selection of different TCP modes of operation such as "no delay", etc.

NORM_CMD(FLUSH) Message Format

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    version    |   type = 3    |           sequence            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           source_id                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     grtt      |     gsize     | flavor = 1 |      flags       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      object_transport_id      |    fec_block_number (msb)     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     fec_block_number (lsb)    |        fec_symbol_id          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

In addition to the NORM common message header and standard NORM_CMD fields, the NORM_CMD(FLUSH) message contains fields to identify the current status and logical transmit position of the sender.

The "flags" field contains sender status information. A single NORM_CMD(FLUSH) flag is currently defined:

NORM_FLUSH_FLAG_EOT = 0x01

When the NORM_FLUSH_FLAG_EOT flag is set, this indicates the sender is preparing to terminate transmission and will no longer provide response to repair requests. This allows the receiver set to gracefully reach closure of operation with this sender and free any resources that are no longer needed.

The "object_transport_id", "fec_block_number", and "fec_symbol_id" fields indicate the sender's current logical "transmit position". These fields are interpreted in the same manner as the fields of the same names in the NORM_DATA message type. Upon receipt of the the NORM_CMD(FLUSH), receivers are expected to check their completion state _through_ (including) this transmission position. If receivers have outstanding repair needs in this range, they SHALL initiate the NORM NACK Repair Process as described in Section 5.3. If receivers have no outstanding repair needs, no response is generated.

For NORM_OBJECT_STREAM objects, receivers MUST request "explicit-only" repair of the identified "fec_block_number" if the given "fec_symbol_id" is less than the sender's "fec_max_block_len - 1". This condition indicates the sender has not yet completed encoding the corresponding FEC block and parity content is not yet available. An "explicit-only" repair request consists of NACK content for the applicable "fec_block_number" which does not include any requests for parity-based repair. This allows NORM sender applications to "flush" an ongoing stream of transmission when needed, even if in the middle of an FEC block. Once the sender resumes stream transmission and passes the end of the pending coding block, subsequent NACKs from receivers SHALL request parity-based repair as normal. Note that the use of a systematic FEC code is assumed here. Normal receiver NACK inititation and construction is discussed in detail in Section 5.3.

NORM_CMD(SQUELCH) Message

The NORM_CMD(SQUELCH) command is transmitted in response to invalid NORM_NACK content received by the sender. Invalid NORM_NACK content consists of repair requests for NormObjects for which the sender is unable or unwilling to provide repair. This includes repair requests for outdated objects, aborted objects, or those objects which the sender previously transmitted marked with the NORM_FLAG_UNRELIABLE flag. This command indicates to receivers what content is available for repair, thus serving as a description of the sender's current "repair window". Receivers SHALL not generate repair requests for content identified as invalid by a NORM_CMD(SQUELCH).

The NORM_CMD(SQUELCH) command is sent once per 2*GRTT at the most. The NORM_CMD(SQUELCH) advertises the current "repair window" of the sender by identifying the earliest (lowest) transmission point for which it will provide repair, along with an encoded list of objects from that point forward that are no longer valid for repair. This mechanism allows the sender application to cancel or abort transmission and/or repair of previously enqueued objects. The list also contains the identifiers for any objects within the repair window which were sent with the NORM_FLAG_UNRELIABLE flag set. In normal conditions, it is expected the NORM_CMD(SQUELCH) will be needed infrequently, and generally only to provide a reference repair window for receivers who have fallen "out-of-sync" with the sender due to extremely poor network conditions.

The starting point of the invalid NormObject list begins with the lowest invalid NormTransportId greater than the current "repair window" start from the invalid NACK(s) that prompted the generation of the squelch. The length of the list is limited by the sender's NormSegmentSize. This allows the receivers to learn the status of the sender's applicable object repair window with minimal transmission of NORM_CMD(SQUELCH) commands. The format of the NORM_CMD(SQUELCH) message is:

NORM_CMD(SQUELCH) Message Format

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    version    |   type = 3    |          sequence             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           source_id                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      grtt     |     gsize     |  flavor = 2   |    reserved   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      object_transport_id      |    fec_block_number (msb)     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     fec_block_number (lsb)    |        fec_symbol_id          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        invalid_object_list  ...               |
```

In addition to the NORM common message header and standard NORM_CMD fields, the NORM_CMD(SQUELCH) message contains fields to identify the earliest logical transmit position of the sender's current repair window and an "invalid object list" beginning with the index of the logically earliest invalid repair request from the offending NACK message which initiated the squelch transmission.

The "object_transport_id", "fec_block_number", and "fec_symbol_id" fields are concatenated to indicate the beginning of the sender's current repair window (i.e. the logically earliest point in its transmission history for which the sender can provide repair). This serves as an advertisement of a "synchronization point" for receivers to request repair. Note, that while the "fec_symbol_id" is provided here, the sender's repair window will generally be incremented on an FEC coding block basis and the "fec_symbol_id" will be zero.

The "invalid_object_list" is a list of 16-bit NormTransportIds that, although they are within the sender's current repair window, are no longer available for repair from the sender. For example, a sender application may dequeue an out-of-date object even though it is still within the repair window. The total size of the "invalid_object_list" content is implied by the packets payload length and is limited to a maximum of the NormSegmentSize of the sender. Thus, for very large repair windows, it is possible that a single NORM_CMD(SQUELCH) message may not be capable of listing the entire set of invalid objects in the repair window. In this case, the sender SHALL ensure that the list begins with a NormObjectId that is greater than or equal to the lowest ordinal invalid NormObjectId from the NACK message(s) that prompted the NORM_CMD(SQUELCH) generation. The NormObjectIds in the "invalid_object_list" must be greater than the "object_transport_id" marking the beginning of the sender's repair window. This insures convergence of the squelch process, even if multiple invalid NACK/ squelch iterations are required. This explicit description of invalid content within the sender's current window allows the sender application (most notably for discrete "object" based transport) to arbitrarily invalidate (i.e. dequeue) portions of enqueued content (e.g. certain objects) for which it no longer wishes to provide reliable transport.

NORM_CMD(REPAIR_ADV) Message

The NORM_CMD(REPAIR_ADV) message is used by the sender to "advertise" its aggregated repair state from accumulated NORM_NACK messages accumulated during a repair cycle and/or congestion control feedback received. This message is sent only when the sender has received NORM_NACK and/or NORM_ACK(RTT) (when congestion control is enabled) messages via unicast transmission instead of multicast. By "echoing" this information to the receiver set, suppression of feedback can be achieved even when receivers are unicasting that feedback instead of multicasting it among the group[11].

NORM_CMD(REPAIR_ADV) Message Format

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    version    |   type = 3    |          sequence             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          source_id                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      grtt     |     gsize     |  flavor = 4   |     flags     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    cc_flags   |     cc_rtt    |            cc_rate            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      repair_adv_content ...                   |
```

The "grtt", "gsize" and "flavor" fields serve the same purpose as in other NORM_CMD messages.

The "flags" field provide information on the NORM_CMD(REPAIR_ADV) content. There is currently one NORM_CMD(REPAIR_ADV) flag defined:

NORM_REPAIR_ADV_FLAG_LIMIT = 0x01

This flag is set by the sender when it is unable to fits its full current repair state into a single NormSegmentSize. If this flag is set, receivers should limit their NACKing to generating NACKs only up through the maximum ordinal transmission position (objectId::fecBlockId::fecSymbolId) included in the "repair_adv_content".

When congestion control operation is enabled, the "cc_flags", "cc_rtt", and "cc_rate" fields contain values for the receiver with the lowest calculated congestion control rate from which feedback was received since the last NORM_CMD(REPAIR_ADV) transmission. These fields are used by receivers to suppress rounds of congestion control feedback. The definition of these fields is given in the description of the NORM_CMD(CC) message below.

The "repair_adv_content" is in exactly the same form as the "nack_content" of NORM_NACK messages and can be processed by receivers for suppression purposes in the same manner with the exception of the condition when the NORM_REPAIR_ADV_FLAG_LIMIT is set.

NORM_CMD(CC) Message

The NORM_CMD(CC) messages contains fields to enable sender->receiver group greatest round-trip time (GRTT) measurement and provide congestion control information to the group. The NORM_CMD(CC) message is usually transmitted as part of NORM congestion control operation. If NORM is operated in a private network with congestion control operation disabled, the NORM_CMD(CC) message is then used to facilitate GRTT measurement by the sender.

NORM_CMD(CC) Message Format

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    version    |  type = 3     |           sequence            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           source_id                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     grtt      |     gsize     |  flavor = 5   |     flags     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          send_time_sec                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          send_time_usec                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           send_rate           |  cc_sequence  |    reserved   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          cc_node_list ...                     |
```

The NORM common message header and standard NORM_CMD fields serve their usual purposes.

The "flags" field is used to indicate NORM_CMD(CC) options. Currently a single NORM_CMD(CC) flag is defined:

```
                      NORM_CC_FLAG_ENABLE = 0x01
```

When set, this indicates the sender has enabled congestion control feedback collection, and receivers should respond observing the procedures describe in Section 5.5.2, "NORM Congestion Control Operation". When this flag is cleared (i.e. congestion control feedback collection is disabled), this indicates the sender is not observing congestion control operation and the NORM_CMD(CC) message is being used only to provide a reference timestamp for GRTT measurement via receiver NORM_NACK feedback.

The "send_time" field is a timestamp indicating the time that the NORM_CMD(CC) message was transmitted. This consists of a 64-bit field containing 32-bits with the time in seconds ("sent_time_sec") and 32-bits with the time in microseconds ("send_time_usec") since some reference time the source maintains (usually 00:00:00, 1 January 1970). The byte ordering of the fields is "Big Endian" network order. Receivers use this timestamp adjusted by the amount of delay from when they received the NORM_CMD(CC) message to when they respond for the "grtt_response" portion of NORM_ACK and NORM_NACK messages generated. This allows the sender to evaluate the round-trip time to different receivers for congestion control and other (e.g. GRTT determination) purposes.

The "send_rate" field indicates the sender's current transmission rate in bytes per second. The 16-bit "send_rate" field consists of 12 bits of mantissa in the most significant byte and 4 bits of base 10 exponent (order of magnitude) information in the least significant byte. The 12-bit mantissa portion of the field is scaled such that a floating point value of 0.0 corresponds to 0 and a floating point value of 10.0 corresponds to 4096. Thus:

```
            value = (int) (mantissa * 4096.0 / 10.0 + 0.5)
```

For example, to represent a transmission rate of 256kbps (3.2e+04 bytes per second), the lower 4 bits of the 16-bit field contain a value of 0x04 to represent the exponent while the upper 12 bits contain a value of 0x51f as determined from the equation given above:

```
      value = (int)((3.2 * 4096.0 / 10.0) + 0.5) = 1311 = 0x51f
```

To decode the "send_rate" field, the following equation can be used:

```
 sendRate = <upper12bits> * 10.0 / 4096.0 * power(10.0, <lower4bits>)
```

Note the maximum transmission rate representable by this scheme is approximately 9.99e+15 bytes per second.

The "cc_sequence" field is a sequence number applied by the sender to congestion control command messages. The greatest received "cc_sequence" value is recorded by receivers and fed back to the sender in any NORM_ACK or NORM_NACK messages generated by the receivers for that sender.

The "reserved" field is for potential future use and should be set to zero in this version of the NORM protocol.

The "cc_node_list" consists of a list of NormNodeIds and their associated congestion control status. This includes the current limiting receiver (CLR) node, any potential limiting receiver (PLR) nodes which have been identified, and some number of receivers for which congestion control status is being provided, most notably including the receivers' current RTT measurement. The length of the "cc_node_list" provides for at least the CLR and one other receiver, but may be configurable for more timely feedback to the group. The list length can be inferred from the length of the NORM_CMD(CC) Each item in the "cc_node_list" is in the following format:

Congestion Control Node List Item Fields

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           cc_node_id                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    cc_flags   |     cc_rtt    |            cc_rate            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The "cc_node_id" is the NormNodeId of the receiver which the item represents.

The "cc_flags" field contains flags indicating the congestion control status of the indicated receiver. The following flags are defined:

Flag	Value	Purpose
NORM_CC_FLAG_CLR	0x01	Receiver is the current limiting receiver (CLR)

NORM_CC_FLAG_PLR	0x02	Receiver is a potential limiting receiver (PLR)
NORM_CC_FLAG_RTT	0x04	Receiver has measured RTT with respect to sender
NORM_CC_FLAG_START	0x08	Sender/receiver is in "slow start" phase of congestion control operation (i.e. The receiver has not yet detected any packet loss and the "cc_rate" field is a function of the receiver's measured receive rate).
NORM_CC_FLAG_LEAVE	0x10	Receiver is imminently leaving the session and its feedback should not be considered in congestion control operation.

The "cc_rtt" contains a quantized representation of the receiver's individual sender<->receiver RTT as measured by the sender. This field is valid only if the NORM_FLAG_RTT flag is set in the "cc_flags" field. This one byte field is a quantized representation of the RTT using the algorithm described in the NORM Building Block document [15].

The "cc_rate" field contains a representation of the receiver's current calculated (during steady-state congestion control operation) or twice its measured (during the "slow start" phase) congestion control rate. This field is encoded and decoded using the same technique as described for the NORM_CMD(CC) "send_rate" field.

NORM_CMD(ACK_REQ) Message

The NORM_CMD(ACK_REQ) message is used by the sender to request acknowledgement from a specified list of receivers. This message is used in providing a lightweight positive acknowledgement mechanism that is OPTIONAL for use by the reliable multicast application. The NORM protocol defines a specific acknowledgement mechanism to determine that watermark points in the reliable transmission have been achieved by specific receivers. Addtionally, a range of acknowledgement request types is provided for use at the application's discretion. Provision for application-defined, positively-acknowledged commands allows the application to automatically take advantage of transmission and round-trip timing information available to the NORM protocol. The details of the NORM positive acknowledgement process including transmission of the NORM_CMD(ACK_REQ) messages and the receiver response (NORM_ACK) are described in Section 5.5.3. The format of the NORM_CMD(ACK_REQ) message is:

NORM_CMD(ACK_REQ) Message Format

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    version    |   type = 3    |           sequence            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            source_id                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      grtt     |     gsize     |  flavor = 3   |    ack_type   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         ack_req_content                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     ack_req_content (cont'd)                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        acking_node_list ...                   |
```

The NORM common message header and standard NORM_CMD fields serve their usual purposes.

The "ack_type" field indicates type of acknowledemgent being requested and thus implies rules for how the receiver will treat this request. The following "ack_type" values are defined and are also used in NORM_ACK messages described later:

ACK Type	Value	Purpose
NORM_ACK(WATERMARK)	1	Used to request acknowledgement of reliable reception of watermark transmission point.
NORM_ACK(CC)	2	Used to identify NORM_ACK messages sent for congestion control only.
NORM_ACK(RESERVED)	3-15	Reserved for possible future NORM protocol use.
NORM_ACK(APPLICATION)	16-255	Used at application's discretion.

The "ack_req_content" field consists of 8 bytes which is interpreted differently for different "ack_type" values.

The "acking_node_list" field is a list of NormNodeIds. The listed NormNodes are expected to explicitly respond to the acknowledgement request according the rules for the type of acknowledgment requested and the NORM Positive Acknowledgment procedure described in Section 5.3.3.

The NORM_ACK(WATERMARK) type indicates the sender wishes to receive acknowledgement from receivers in the "acking_node_list" who have achieved completion of reception through a specific "watermark point" in terms of a logical transmission position. This "watermark point" is given in the "ack_req_content" field.

The format of the NORM_CMD(ACK_REQ(WATERMARK)) message is:

NORM_CMD(ACK_REQ(WATERMARK)) Message Format

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    version    |   type = 3    |           sequence            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           source_id                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      grtt     |     gsize     |  flavor = 3   | ack_flavor = 1|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      object_transport_id      |    fec_block_number (msb)     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     fec_block_number (lsb)    |        fec_symbol_id          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      acking_node_list ...                     |
```

The NORM common message header and standard NORM_CMD fields serve their usual purposes. The "ack_flavor" is set to a value of one.

The "object_transport_id", "fec_block_number", and "fec_symbol_id" are used to identify the watermark point for which positive acknowledgement is requested. This watermark point is similar to the transmission position given in NORM_CMD(FLUSH) messages. Furthermore, NORM receivers (whether or not they are included in the "acking_node_list") SHALL treat the ACK_REQ(WATERMARK) command as equivalent to a NORM_CMD(FLUSH) command and appropriately initiate NACK repair cycles in response to any detected missing data up through the indicated watermark point.

The "acking_node_list" field contains the NormNodeIds of the current NORM receivers which should positive acknowledge (NORM_ACK) this request. The packet payload length implies the length of the "acking_node_list" and its length is limited to the NormSegmentSize. The individual NormNodeId items are listed in network (Big Endian) order. If a receiver is included in the "acking_node_list" and it has no repair needs up through the watermark point, it SHALL schedule transmission of a NORM_ACK message as described in Section 5.5.3.

The NORM_ACK(CC) type is provided only for when receivers generate NORM_ACK messages in response to NORM_CMD(CC) messages for congestion control operation. There is no corresponding NORM_CMD(ACK_REQ(CC)) message.

The NORM_ACK(RESERVED) range of types is provided for possible future NORM protocol use.

The NORM_ACK(APPLICATION) range of types is provided so that NORM applications may implement application-defined, positively-acknowledged commands which are able to leverage internal transmission and round-trip timing information available to the NORM protocol implementation. The interpretation of the "ack_req_content" is application-defined in this case.

NORM_CMD(APPLICATION) Message

This command allows the NORM application to robustly transmit application-defined commands. The command message preempts any ongoing data transmission and is repeated NORM_ROBUST_FACTOR times at a rate of once per 2*GRTT. This rate of repetition allows the application to collect any response (if that is the application's purpose for the command) before it is repeated. Possible responses might include initiation of data transmission , NORM_CMD(APPLICATION) messages, or even application-defined, positively-acknowledge commands from other NormSession participants. The transmission of these commands will preempt data transmission when they are scheduled and may be multiplexed with ongoing data transmission. This type of robustly transmitted command allows NORM applications to define a complete set of session control mechanisms with less state than the transfer of FEC encoded reliable content requires while taking advantage of NORM transmission and round-trip timing information.

```
NORM_CMD(APPLICATION) Message Format
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    version    |  type = 3     |          sequence             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           source_id                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      grtt     |     gsize     |  flavor = 6   |   reserved    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   application defined content ...             |
```

The NORM common message header and NORM_CMD fields are interpreted as previously described.

The "application-defined content" contains information in a format at the discretion of the application. The size of this payload is limited a maximum of the sender's NormSegmentSize setting.

4.3 Receiver Messages

The NORM message types generated by pariticipating receivers consist of NORM_NACK and NORM_ACK message types. NORM_NACK messages are sent to request repair of missing data content from sender transmission and NORM_ACK messages are generated in response to certain sender commands including NORM_CMD(CC) and NORM_CMD(ACK_REQ).

4.3.1 NORM_NACK Message

The principal purpose of NORM_NACK messages is for receivers to request repair of sender content via selective, negative acknowledgement upon detection of incomplete data. NORM_NACK messages will be transmitted according to the rules of NORM_NACK generation and suppression described in Section 5.3. The content of these messages is in a format that can potentially be used by compatible intermediate systems [12] to provide assistance in promoting protocol scalability and efficiency when available. NORM_NACK messages also contain additional fields to provide feedback to the sender(s) for purposes of round-trip timing collection and congestion control.

The payload of NORM_NACK messages contains one or more repair requests for different objects or portions of those objects. The NORM_NACK message format is as follows:

NORM_NACK Message Format

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    version    |  type = 3     |           sequence            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            source_id                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            server_id                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        grtt_response_sec                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        grtt_response_usec                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    cc_flags   |     cc_rtt    |            cc_loss            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           cc_rate             | cc_sequence  |    reserved    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           nack_content ...                    |
```

The NORM common message header fields serve their usual purposes.

The "server_id" field identifies the NORM sender to which the NORM_NACK message is destined.

The "grtt_response" fields contain an adjusted version of the timestamp from the most recently received NORM_CMD(CC) message for the indicated NORM sender. The format of the "grtt_response" is the same as the "send_time" field of the NORM_CMD(CC). The "grtt_response" value is _relative_ to the "send_time" the source provided with a corresponding NORM_CMD(CC) command. The receiver adjusts the source's NORM_CMD(CC) "send_time" timestamp by adding the time differential from when the receiver received the NORM_CMD(CC) to when the NORM_NACK is transmitted to calculate the value in the "grtt_response" field. This is the "receive_to_response_differential" value used in the following formula:

```
"grtt_response" = NORM_CMD(CC) "send_time" + receive_to_response_differential
```

The receiver SHALL set the "grtt_response" to a ZERO value, to indicate that the it has not yet received a NORM_CMD(CC) message from the indicated sender and that the sender should ignore the "grtt_response" in this message.

The "cc_flags" field contains bits representing the receiver's state with respect to congestion control operation. The possible values for the "cc_flags" field are those specified for the NORM_CMD(CC) message node list item flags.

The "cc_rtt" field SHALL be set to a default maximum value and the NORM_CC_FLAG_RTT flag SHALL be cleared when the receiver has not yet received RTT measurement information. When the receiver has received RTT measurement information, it shall set the "cc_rtt" value accordingly and set the NORM_CC_FLAG_RTT flag in the "cc_flags" field.

The "cc_loss" field is the receiver’s current packet loss fraction estimate for the indicated source. The loss fraction is a value from 0.0 to 1.0 corresponding to a range of zero to 100 percent packet loss. The 16-bit "cc_loss" value is calculated by the following formula:

```
             "cc_loss" = decimal_loss_fraction * 65535.0
```

The "cc_rate" field represents the receivers current local congestion control rate. During "slow start", when the receiver has detected no loss, this value is set to twice the actual rate it has measured from the corresponding sender and the NORM_CC_FLAG_START is set in the "cc_flags’ field. Otherwise, the receiver calculates a congestion control rate based on its loss measurement and RTT measurement information (even if default) for the "cc_rate" field.

The "cc_sequence" field contains the current greatest "cc_sequence" number of received NORM_CMD(CC) messages from the corresponding sender. This information can assist the sender in congestion control operation by providing an indicator of how current ("fresh") the receiver’s round-trip measurement reference time is and whether the receiver has been successfully receiving recent congestion control probes. For example, if it is apparent the receiver has not been receiving recent congestion control probes (and thus possibly other messages from the sender), the sender may choose to take congestion avoidance measures.

The "reserved" field is for potential future NORM use and SHALL be set to ZERO for this version of the protocol.

The "nack_content" of the NORM_NACK message specifies the repair needs of the receiver with respect to the NORM sender indicated by the "server_id" field. The receiver constructs repair requests based on the NORM_DATA and/or NORM_INFO segments it requires from the sender in order to complete reliable reception. A single repair request consists of a list of items, ranges, and/or FEC coding block erasure counts for needed NORM_DATA and/or NORM_INFO content. Multiple repair requests may be concatenated within the "nack_content" field of a NORM_NACK message. Note that a single repair request can possibly include multiple "items", "ranges", or "erasure_counts". In turn, the "nack_content" field may contain multiple repair request. A single repair request has the following format:

NACK Repair Request Format

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    form       |    flags      |            length             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     object_transport_id       |     fec_block_number (msb)    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     fec_block_number (lsb)    | fec_symbol_id or erasure_count|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              ...                              |
```

The "form" field indicates currently whether the repair request content that follows is a list of NORM_NACK_ITEMS, NORM_NACK_RANGES, or NORM_NACK_ERASURES. Possible values for the "form" field include:

Form	Value
NORM_NACK_ITEMS	1
NORM_NACK_RANGES	2
NORM_NACK_ERASURES	3

When the repair request consists of individual NORM_NACK_ITEMS, each concatenation of object_transport_id::fec_block_number::fec_symbol_id identifies an individual repair need. When the repair request "form" is NORM_NACK_RANGES, the inclusive range of sender information needed by the receive is given in pairs of object_transport_id::fec_block_number::fec_symbol_id. When the repair request form is NORM_NACK_ERASURES, each object_transport_id::fec_block_number::erasure_count concatenation listed indicates the receiver's FEC erasure count for the identified object and FEC encoding block.

The "flags" field is currently used to indicate if the NACK content applies to NORM_DATA content, NORM_INFO content, or both. Thus, defined flags in this field include:

Flag	Value	Purpose
NORM_NACK_SEGMENT	0x01	Indicates the listed segment(s) are required as repair.
NORM_NACK_BLOCK	0x02	Indicates the entire listed block(s) are required as repair.
NORM_NACK_INFO	0x04	Indicates the object's NORM_INFO is required as repair.
NORM_NACK_OBJECT	0x08	Indicates the entire listed object(s) are required as repair.

When the NORM_FLAG_SEGMENT flag is set, the "object_transport_id", "fec_block_number" and "fec_symbol_id" fields are concatenated to determine which sets or ranges of individual NORM_DATA segments are needed to repair complete content at this receiver. When the NORM_FLAG_BLOCK flag is set, this indicates the receiver is completely missing the indicated coding block(s) and requires transmissions sufficient to repair the indicated block(s) in their entirety. In this case the "fec_symbol_id" repair request fields are ignored. When the NORM_NACK_INFO flag is set, this indicates the receiver is missing the NORM_INFO segment for the indicated "object_transport_id". Note the NORM_NACK_INFO may be set in combination with the NORM_NACK_BLOCK or NORM_NACK_SEGMENT flags, or may be set alone. When the

NORM_NACK_OBJECT flag is set, this indicates the receiver is missing the entire NormTransportObject referenced by the "object_transport_id". This also implicitly requests any available NORM_INFO for the NormObject, if applicable. The "fec_block_number" and "fec_symbol_id" fields are ignored when the flag NORM_NACK_OBJECT is set.

The "length" field is given (in bytes) to indicate the length of the list of repair request items or ranges. Multiple lists of repair request items and/or ranges may be concatenated together within a single NORM_NACK message.

The "object_transport_id", "fec_block_number, and "fec_symbol_id" fields comprise repair request list items to be interpreted according to the repair request "form" and "flags" fields. As noted, when the "form" is NORM_NACK_RANGES, pairs of object_transport_id::fec_block_number::erasure_count define each repair request list item.

NORM_NACK Content Examples:

In Example 1, a list of individual NORM_NACK_ITEM repair requests is given. In Example 2, a list of NORM_NACK_RANGE requests _and_ a single NORM_NACK_ITEM request are concatenated to illustrate the possible content of a NORM_NACK message. Note that FEC coding block erasure counts are provided in each case. The erasure counts are not really necessary since the sender can easily determine the erasure count while processing the NACK content. However, the erasure count option may be useful for operation with Generic Router Assist (GRA).

Example 1:
NORM_NACK content for: Object 12, Coding Block 3, Segments 2,5,8

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   form = 3    | flags = 0x01  |       length  = 8             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   object_transport_id = 12    |   fec_block_number (msb) = 0  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   fec_block_number(lsb) = 3   |    erasure_count  = 3         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   form = 1    | flags = 0x01  |       length  = 24            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   object_transport_id = 12    |   fec_block_number (msb) = 0  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   fec_block_number (lsb) = 3  |    fec_symbol_id  = 2         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   object_transport_id = 12    |   fec_block_number (msb) = 0  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   fec_block_number (lsb) = 3  |    fec_symbol_id  = 5         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   object_transport_id = 12    |   fec_block_number (msb) = 0  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   fec_block_number (lsb) = 3  |    fec_symbol_id  = 8         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Example 2:
NORM_NACK content for: Object 18 Coding Block 6, Segments 5, 6, 7, 8, 9, 10; and Object 19 NORM_INFO and Coding Block 1, segment 3

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   form = 3    | flags = 0x01  |       length  = 8             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   object_transport_id = 18    |  fec_block_number (msb) = 0   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   fec_block_number (lsb) = 6  |    erasure_count  = 6         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   form = 2    | flags = 0x01  |       length  = 16            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   object_transport_id = 18    |  fec_block_number (msb) = 0   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   fec_block_number (lsb) = 6  |    fec_symbol_id  = 5         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   object_transport_id = 18    |  fec_block_number (msb) = 0   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   fec_block_number (lsb) = 6  |    fec_symbol_id  = 10        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   form = 3    | flags = 0x05  |       length  = 8             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   object_transport_id = 19    |  fec_block_number (msb) = 0   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   fec_block_number (lsb) = 1  |    erasure_count  = 1         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   form = 1    | flags = 0x05  |        length  = 8            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   object_transport_id = 19    |  fec_block_number (msb) = 0   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   fec_block_number (lsb) = 1  |    fec_symbol_id  = 3         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

4.3.2 NORM_ACK Message

The NORM_ACK message is primarily used as part of NORM congestion control operation and round-trip timing measurement. The generation of NORM_ACK messages for round-trip timing and congestion-control operation is described in Sections 5.5.1 and 5.5.2, respecctively. Some applications may benefit from some limited form of positive acknowledgement for certain functions. A simple, scalable positive acknowledgement scheme is defined in Section 5.5.3 which can be leveraged by protocol implementations when appropriate.

NORM_ACK Message Format

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    version    |  type = 3     |           sequence            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           source_id                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           server_id                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       grtt_response_sec                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       grtt_response_usec                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    cc_flags   |     cc_rtt    |            cc_loss            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            cc_rate            |  cc_sequence  |    ack_type   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         ack_content ...                       |
```

The NORM common message header fields serve their usual purposes.

The "server_id", "grtt_response", "cc_flags", "cc_rtt", "cc_loss", "cc_rate", and "cc_sequence" fields serve the same purpose as the corresponding fields in NORM_NACK messages.

The "ack_type" field indicates the nature of the NORM_ACK message. This directly corresponds to the "ack_type" field of the NORM_CMD(ACK_REQ) message.

The "ack_content" format is a function of the "ack_type". The NORM_ACK(CC) message has no attached content. Only the NORM_ACK header applies. In the case of NORM_ACK(WATERMARK), a specific "ack_content" format is defined:

NORM_ACK(WATERMARK) Ack Content

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      object_transport_id      |    fec_block_number (msb)     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     fec_block_number (lsb)    |        fec_symbol_id          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The "object_transport_id", "fec_block_number", and "fec_symbol_id" are used by the receiver to acknowledge a NORM_CMD(ACK_REQ(WATERMARK)) transmitted by the sender identified by the "server_id" field.

The "ack_content" of NORM_ACK messages for application-defined "ack_type" values is specific to the application but is limited in size to a maximum the NormSegmentSize of the sender referenced by the "server_id".

4.4 General Messages

4.4.1 NORM_REPORT

This is an optional message generated by NORM participants. This message could be used for periodic performance reports from receivers in experimental NORM implementations. The format of this message is currently undefined. Experimental NORM implementations may define NORM_REPORT formats as needed for test purposes.

5.0 Functionality Definition

This section describes the detailed interactions of senders and receivers participating in a NORM session. A simple synopsis of protocol operation is given in the following items.

1) The sender periodically transmits NORM_CMD(CC) messages as needed to initialize and collect roundtrip timing and congestion control feedback from the receiver set.

2) The sender transmits an ordinal set of NormObjects segmented in the form of NORM_DATA (and optional NORM_INFO) messages labeled with NormTransportIds and logically identified with FEC encoding block numbers and symbol identifiers.

3) As receivers detect missing content from the sender, they initiate repair requests with NORM_NACK messages. Note the receivers track the sender's most recent object_transport_id::fec_block_number::fec_symbol_id transmit position and NACK _only_ for content ordinally prior to that transmit position. The receivers use random backoff timeouts before generating NORM_NACK messages and wait an appropriate amount of time before repeating the NORM_NACK if their repair request is not satisified.

4) The sender aggregates repair requests from the receiver set and logically "rewinds" to send appropriate repair messages. The sender sends repairs for the earliest ordinal transmit position first and maintains this ordinal repair transmission sequence. Previously untransmitted FEC parity content for the applicable FEC coding block is used for repair transmissions to the greatest extent possible. If the sender exhausts its available FEC parity content on multiple repair cycles for the same coding block, it resorts to an explicit repair strategy (again using parity content) to complete repairs. (The use of explicit repair is expected to be an exception in general protocol operation, but the possibility does exist for extreme conditions). The sender immediately assumes transmission of new content once it has sent pending repair transmissions.

5) The sender transmits NORM_CMD(FLUSH) messages when it reaches the end of newly available transmit content. Receivers respond to the NORM_CMD(FLUSH) messages with NORM_NACK transmissions (following the same suppression backoff timeout strategy as for data) if they require further repair.

6) The sender transmission rate is subject to rate control limits determined by congestion control. Each sender in a NormSession maintains its own independent congestion control state. Receivers provide congestion control feedback in NORM_NACK and NORM_ACK messages. This feedback is

controlled using suppression mechanism similar to that for NORM_NACK messages.

While the overall concept of the protocol is relatively simple, there are details to each of these aspects that need to be addressed for successful, robust, and scalable operation.

5.1 NORM Sender Initialization and Transmission

Upon startup, the NORM sender immediately begins sending NORM_CMD(CC) messages to collect GRTT and other information from the potential group. If congestion control operation is enabled the NORM_CC_FLAG_ENABLE MUST be set. Congestion control operation SHALL be observed at all times when operating in the general Internet. Even if congestion control operation is disabled at the sender, it may be desirable to set the the NORM_CC_FLAG_ENABLE to proactively collect feedback from the receivers to have input to GRTT measurement prior to NACK initiation.

In some cases, applications may wish for the sender to also proceed with data transmission immediately. In other cases, the sender may wish to defer data transmission until it has received some feedback or request from the receiver set indicating that receivers are indeed present. Note, in some applications (e.g. web push), this indication may come out-of-band with respect to the multicast session via other means. The periodic transmission of NORM_CMD(CC) messages may precede actual data transmission in order to have initial GRTT measurement.

The NORM protocol sender message headers contain all information necessary to configure receivers for subsequent reliable reception. This includes FEC coding parameters, the sender NormSegmentSize, and other information. Additionally, applications may leverage the use of NORM_INFO messages associated with the session data objects in the session to provide application-specific context information for the session and data being transmitted.

The NORM sender begins segmenting application-enqueued data into NORM_DATA segments and transmitting it to the group. The rate of transmission is controlled via the congestion control mechanisms described in this document or at a fixed rate if desired for closed network operations. The receivers participating in the multicast group provide feedback to the sender as needed. When the sender reaches the end of data it has enqueued for transmission or any pending repairs, it transmits a series of NORM_CMD(FLUSH) messages at a rate of one per 2*GRTT. Receivers may respond to these NORM_CMD(FLUSH) messages with additional repair requests. A protocol parameter "NORM_ROBUST_FACTOR" determines the number of flush messages sent. If receivers request repair, the repair is provided and flushing occurs again at the end of repair transmission.

5.2 NORM Receiver Initialization and Reception

The NORM protocol is designed such that receivers may join and leave the group at will. However, some applications may be constrained such that receivers need to be members of the group prior to start of data transmission. NORM applications may use different policies to constrain the impact of new receivers joining the group in the middle of a session. For example, a useful implementation policy is for new receivers joining the group to restrain requesting repair of transport objects in progress. The NORM sender implementation may wish to impose additional constraints to limit the ability of receivers to disrupt reliable multicast performance by joining, leaving, and rejoining the group often. Different receiver "join policies" may be appropriate for different applications and/or scenarios. A default policy of allowing receivers to request repair only for coding blocks with a NormTransportId and FEC coding block number greater than or equal to the first non-repair NORM_DATA or NORM_INFO message received upon joining the group is RECOMMENDED for general purpose operation.

5.3 NORM Receiver NACK Procedure

When the receiver detects it is missing data from a sender's NORM transmissions, it initiates its NACKing procedure. The NACKing procedure SHALL be initiated _only_ at NormObject boundaries, FEC coding block boundaries, or upon receipt of a NORM_CMD(FLUSH) or NORM_CMD(ACK_REQ(WATERMARK)) message.

The NACKing procedure begins with a random backoff timeout. The duration of the backoff timeout is chosen using the "RandomBackoff" algorithm described in the NORM Building Block document [15] using (K*GRTTsender) for the "maxTime" parameter and the sender advertised group size (GSIZEsender) as the "groupSize" parameter. The backoff factor "K" MUST be greater than one to provide for feedback suppression. A value of K = 4 is RECOMMENDED for the Any Source Multicast (ASM) model while a value of K = 6 is RECOMMENDED for Single Source Multicast (SSM) operation. Thus:

```
T_backoff = RandomBackoff(K*GRTTsender, GSIZEsender)
```

During this backoff time, the receiver accumulates external pending repair state from NORM_NACK messages and NORM_CMD(REPAIR_ADV) messages received. At the end of the backoff time, the receiver SHALL generate a NORM_NACK message only if the following conditions are met:

1) The sender's current transmit position (in terms of object_transport_id::fec_block_number::fec_symbol_id) exceeds the earliest repair position of the receiver.

2) The repair state accumulated from NORM_NACK and NORM_CMD(REPAIR_ADV) messages do not equal or supersede the receiver's repair needs.

If these conditions are met, the receiver immediately generates a NORM_NACK message when the backoff timeout expires.

The content of the NORM_NACK message contains repair request content beginning with lowest ordinal repair position for the receiver up to the most recently heard ordinal transmission position for the sender. If the size of the NORM_NACK content exceeds the NormSegmentSize, the NACK content is limited to that point so that the receiver only generates a single NORM_NACK message per NACK cycle for a given sender.

For each partially-received FEC coding block requiring repair, the receiver SHALL, on its _first_ repair attempt for the block, request the parity portion of the FEC coding block beginning with the lowest ordinal _parity_ "fec_symbol_id" and request the number of symbols corresponding to its data segment erasure count for the block. On _subsequent_ repair cycles for the same coding block, the receiver SHALL request only those repair symbols from the first set it has not yet received up to the remaining erasure count for that applicable coding block. Note that the sender may have provided other additional parity segments for other receivers that could also be used to satisfy the local receiver's erasure-filling needs. In the case where the erasure count for a partially-received FEC coding block exceeds the maximum number of parity symbols available from the sender for the block (as indicated by the NORM_DATA "fec_num_parity" field), the receiver SHALL request all available parity segments and the ordinally highest missing data segments required to satisfy its erasure needs for the block. The goal of this strategy is for the overall receiver set to request a lowest common denominator set of repair symbols for a given FEC coding block. This allows the sender to construct the most efficient repair transmission segment set and enables effective NACK suppression among the receivers even with uncorrelated packet loss. This approach also requires no synchronization among the receiver set in their repair requests for the sender.

For FEC coding blocks or NormObjects missing in their entirety, the NORM receiver constructs repair requests with NORM_NACK_BLOCK or NORM_NACK_OBJECT flags set as appropriate. The request for retransmission of NORM_INFO is accomplished by setting the NORM_NACK_INFO flag in a corresponding repair request.

5.4 NORM Sender NACK Processing and Repair Transmission

The principle goal of the sender is to make forward progress in the transmission of data its application has enqueued. However, the sender must occasionally "rewind" to satisfy the repair needs of receivers who have NACKed. Aggregation of multiple NACKs is used to determine an optimal repair strategy when a NACK event occurs. Since receivers initiate the NACK process on coding block or object boundaries, there is some loose degree of synchronization of the repair process.

5.4.1 NORM Sender Repair State Aggregation

When a sender is in its normal state of transmitting new data and receives a NACK, it begins a procedure to accumulate NACK repair state from NORM_NACK messages before beginning repair transmissions. Note that this period of aggregating repair state does _not_ interfere with its ongoing transmission of new data.

The period of time during which the sender aggregates NORM_NACK messages is equal to K*GRTT where "K" is the same backoff scaling value used by the receivers and "GRTT" is the sender's current estimate of the group's greatest round-trip time. When this period ends, the sender "rewinds" by incorporating the accumulated repair state into its pending transmission state and begins transmitting repair messages, then continues with new transmissions of any enqueued data. Also, at this point in time, the sender begins a "holdoff" timeout of 1*GRTT during which time the sender constrains itself from initiating a new repair aggregation cycle, even if NORM_NACK messages arrive. If additional NORM_NACK messages are received during this hold-off period, the sender will immediately incorporate these "late messages" into its pending transmission state ONLY if the NACK content is ordinally greater than the sender's current transmission position. This "holdoff" time allows worst case time for the sender to propagate its current transmission sequence position to the group, thus avoiding redundant repair transmissions. After the holdoff timeout expires, a new NACK accumulation period can be begun (upon arrival of a NACK) in concert with the pending repair and new data transmission. The sender repeats the same process of incorporating accumulated repair state into its transmission plan during the the new aggregation period and subsequently "rewinding" to transmit the lowest ordinal repair data.

5.4.2 NORM Sender FEC Repair Transmission Strategy

The NORM sender should leverage transmission of FEC parity content for repair to the greatest extent possible. Recall that the receivers use a strategy to request a lowest common denominator of explicit repair (including parity content) in the formation of their NORM_NACK messages. Before falling back to explicitly satisfying all of the different receivers' repair needs, the sender can make use of the general erasure-filling capability of FEC-generated parity segments. The sender can determine the maximum erasure filling needs for individual FEC coding blocks from the NORM_NACK messages received during the repair aggregation period. Then, if the sender has a sufficient number (less than or equal to the maximum erasure count) of previously unsent parity segments available for the applicable coding blocks, the sender can transmit these in lieu of the specific packets the receiver set has requested. Only after exhausting its supply of "fresh" (unsent) parity segments for a given coding block should the sender resort to explicit transmission of the receiver set's repair needs. In general, if a sufficiently powerful FEC code is used, the need for explicit repair will be an exception, and the fulfillment of reliable multicast can be accomplished quite efficiently. However, the ability to resort to explicit repair allows the protocol to be reliable under even very extreme circumstances.

NORM_DATA messages sent as repair transmissions are flagged with the NORM_FLAG_REPAIR flag. This allows receivers to obey any policies that limit new receivers from joining the reliable transmission on repair transmissions.

To facilitate operation with Generic Router Assist (GRA) [12], the sender can additionally flag NORM_DATA transmissions sent as explicit repair with the NORM_FLAG_EXPLICIT flag. The GRA router needs to only subcast a sufficient count of non-explicit parity repairs to satisfy the sub-tree's erasure filling needs for a given FEC coding block. When the sender has resorted to explicit repair, the GRA router will subcast all of the explicit repair packets to those portions of the routing tree still requiring repair for a given coding block. (Note the GRA router will be required to conduct repair state accumulation for sub-routes in a manner similar to the sender's repair state accumulation in order to have sufficient information to perform the subcasting. Additionally, the GRA router can perform additional NORM_NACK suppression/aggregation as it conducts this repair state accumulation for NORM repair cycles).

5.4.3 NORM Sender NORM_CMD(SQUELCH) Generation

If the sender receives a NORM_NACK message for repair of data it is no longer supporting, the sender generates a NORM_CMD(SQUELCH) message to advertise its repair window and squelch any receivers from additional NACKing of invalid data. The transmission rate of NORM_CMD(SQUELCH) messages is limited to once per 2*GRTT. The "invalid_object_list" (if applicable) of the NORM_CMD(SQUELCH) message SHALL begin with the lowest "object_transport_id" from the invalid NORM_NACK messages received since the last NORM_CMD(SQUELCH) transmission. Lower ordinal invalid "object_transport_ids" should be included only while the NORM_CMD(SQUELCH) payload is less than the sender’s NormSegmentSize parameter.

5.4.4 NORM Sender NORM_CMD(REPAIR_ADV) Generation

When a NORM sender receives NORM_NACK messages from receivers via unicast transmission, it uses NORM_CMD(REPAIR_ADV) messages to advertise its accumulated repair state to the receiver set since the receiver set is not directly sharing their repair needs via multicast communication. The NORM_CMD(REPAIR_ADV) message is multicast to the receiver set by the sender. The payload portion of this message has content in the same format as the NORM_NACK receiver message payload. Receivers are then able to perform feedback suppression in the same manner as with NORM_NACK messages directly received from other receivers. Note the sender does not merely retransmit NACK content it receives, but instead transmits a representation of its aggregated repair state. The transmission of NORM_CMD(REPAIR_ADV) messages are subject to the sender transmit rate limit and NormSegmentSize limitation. When the NORM_CMD(REPAIR_ADV) message is of maximum size, receivers SHALL consider the maximum ordinal transmission position value embedded in the message as the senders "current" transmission position and suppress requests for ordinally higher repair. For congestion control operation, the NORM_CMD(REPAIR_ADV) fields of "cc_flags", "cc_rtt", and "cc_rate" contain the "worst case" values received for each field since the last NORM_CMD(REPAIR_ADV) transmission. This means the minimum received "cc_rate" and the set of "cc_flag" values resulting in the most suppression (i.e. the NORM_CC_FLAG_RTT flag is unset if _any_ congestion control feedback was received with that flag unset since the last NORM_CMD(REPAIR_ADV) transmission).

5.5 Additional NORM Protocol Mechanisms

In addition to the principal function of data content transmission and repair, there are some other protocol mechanisms that help NORM to adapt to network conditions and play fairly with other coexistent protocols.

5.5.1 NORM Greatest Round-trip Time (GRTT) Collection

For NORM receivers to appropriately scale backoff timeouts and the senders to use proper corresponding timeouts, the participants must agree on a common timeout basis. Each NORM sender monitors the round-trip time of active receivers and determines the group greatest round-trip time (GRTT). The sender advertises this GRTT estimate in every message it transmits so that receivers have this value available for scaling their timers. To measure the current GRTT, the sender periodically sends NORM_CMD(CC) messages which contain a locally generated timestamp. Receivers are expected to record this timestamp along with the time the NORM_CMD(CC) message is received. Then, when the receivers generate feedback messages to the sender, an adjusted version of the sender timestamp is embedded in the feedback message (NORM_NACK or NORM_ACK). The adjustment adds the amount of time the receiver held the timestamp before generating its response. Upon receipt of this adjusted timestamp, the sender is able to calculate the round-trip time to that receiver.

The round-trip time for each receiver is fed into an algorithm that weights and smooths the values for a conservative estimate of the GRTT. The algorithm and methodology is described in the NORM Building Block document [11] in the section entitled "One-to-Many Sender GRTT Measurement". A conservative estimate helps feedback suppression at a small cost in overall protocol repair delay. The sender's current estimate of GRTT is advertised in the "grtt" field found in all NORM sender messages. The advertised GRTT is also limited to be at least as big as the nominal inter-packet transmission time given the sender's current transmission rate. The reason for this additional limit is to keep the receiver somewhat "event driven" by making sure the sender has had adequate time to generate any response to repair requests from receivers given transmit rate limitations due to congestion control or configuration.

When the NORM_CC_FLAG_ENABLE is set in NORM_CMD(CC) messages, the receivers respond to NORM_CMD(CC) messages as described in Section 5.5.2, "NORM Congestion Control Operation". The NORM_CMD(CC) messages are periodically generated by the sender as described for congestion control operation. This provides for active, but controlled, feedback from the group in the form of NORM_ACK messages and can provide GRTT feedback even if no NORM_NACK messages are being sent. If operating without congestion control in a closed network, the NORM_CMD(CC) messages may be sent periodically with the NORM_CC_FLAG_ENABLE flag cleared. In this case, receivers will only provide GRTT measurement feedback when NORM_NACK messages are generated as no NORM_ACK messages are generated in response to the NORM_CMD(CC). In this case, the NORM_CMD(CC) messages may be sent less frequently, as little as once per minute, to conserve network capacity. Note that the NORM_CC_FLAG_ENABLE can also be set to actively solicit RTT feedback from the receiver group per congestion control operation even though the sender may not be observing congestion control rate adjustment. NORM operation without congestion control should only be considered in closed networks.

5.5.2 NORM Congestion Control Operation

This section describes congestion control operation for the NORM protocol. The supporting NORM message formats and approach described here are an adaptation of the equation-based TCP-Friendly Multicast Congestion Control (TFMCC) approach described in [18] and [21]. With this TFMCC-based approach, the transmission rate of NORM senders is controlled in a rate-based manner as opposed to window-based congestion control algorithms as in TCP. However, it is possible that the NORM protocol message set may alternatively be used to support a window-based multicast congestion control scheme such as PGMCC [22]. The details of that alternative may be described separately or in a future revision of this document. In either case (rate-based TFMCC or window-based PGMCC), successful control of sender transmission depends upon collection of sender->receiver packet loss estimates and sender<->receiver RTT to identify the congestion control bottleneck path(s) within the multicast topology and adjust the sender rate accordingly. The receiver with loss and RTT estimates that correspond to the lowest result transmission rate is identified as the "current limiting receiver" (CLR).

The steady-state sender transmission rate, to be "friendly" with competing TCP flows is calculated as:

```
                                       S
Rsender = --------------------------------------------------------------
          tRTT * (sqrt((2/3)*p) + 12 * sqrt((3/8)*p) * p * (1 + 32*(p^2)))
           where
```

   S = Nominal transmitted packet size. (The "nominal" packet size
       is determined by the sender as an exponentially weighted
       moving average (EWMA) of transmitted packet sizes to account
       for variable message sizes).

tRTT = The RTT estimate of the current "current limiting receiver"
       (CLR).

   p = The loss event fraction of the CLR.

To support congestion control feedback collection and operation, the NORM sender periodically transmits NORM_CMD(CC) command messages. The GRTT is determined from congestion control feedback included in NACKs and ACKs from the receiver set. The NORM_CMD(CC) messages are multiplexed with NORM data and repair transmissions and serve several purposes:

1) Stimulate explicit feedback from the general receiver set to collect congestion control information.

2) Communicate state to the receiver set on the sender's current congestion control status including details of the CLR.

3) Initiate rapid (immediate) feedback from the CLR in order to closely track the dynamics of congestion control for that "worst path" in the sender->receiver multicast topology.

The format of the NORM_CMD(CC) message is describe in Section 4.2.3 of this document. The NORM_CMD(CC) message contains information to allow for determination of sender<->receiver RTTs, to inform the group of the congestion control CLR, and to provide feedback of individual RTT information to receivers in the group. The NORM_CMD(CC) also provides for exciting feedback from a set of potential limiting receiver (PLR) nodes that may be determined administratively or possibly algorithmically based on congestion control feedback. The details of PLR selection are not discussed in this document.

5.5.2.1 NORM_CMD(CC) Transmission

The NORM_CMD(CC) message is tranmitted periodically by the sender along with its normal data transmission. Note that the repeated transmission of NORM_CMD(CC) messages may be initiated some time before transmission of user data content at session startup. This may be done to collect some estimation of the current state of the multicast topology with respect to group and individual RTT and congestion control state.

A NORM_CMD(CC) message is immediately transmitted at sender startup. The interval of subsequent NORM_CMD(CC) message transmission is determined as follows:

1) By default, the interval is set according to the current sender GRTT estimate. A startup GRTT of 0.5 seconds is recommended when no feedback has yet been received from the group.

2) If a CLR has been identified (based on previous receiver feedback), the interval is the sender<->receiver RTT for the CLR.

3) Additionally, if the interval of nominal data message transmission is greater than the GRTT or CLR RTT interval, the NORM_CMD(CC) interval is set to this greater value. This ensures that the transmission of this control message is not done to the exclusion of user data transmission.

The NORM_CMD(CC) "cc_sequence" field is incremented with each transmission of a NORM_CMD(CC) command. The greatest "cc_sequence" recently received by receivers is included in their feedback to the sender. This allows the sender to determine the "age" of feedback to assist in congestion avoidance.

The sender advertises its current transmission rate in the "send_rate" field of the NORM_CMD(CC) message. This rate information is used by receivers to bias the timing of explicit feedback and to initialize loss estimation during congestion control startup or restart.

The "cc_node_list" contains a list of entries identifying receivers and their current congestion control state (status "flags", "rtt" and "loss" estimates). The list may be empty if the sender has not yet received any feedback from the group. If the sender has received feedback, the list will minimally contain an entry identifying the CLR. A NORM_CC_FLAG_CLR flag value is provided for the "cc_flags" field to identify the CLR entry. It is recommended that the CLR entry be the first in the list for implementation efficiency. Additional entries in the list are used to provide sender-measured individual RTT estimates to receivers in the group. The number of additional entries in this list is dependent upon the percentage of control traffic the sender application is willing to send with respect to user data message transmissions. More entries in the list may allow the sender to be more responsive to congestion control dynamics. The length of the list may be dynamically determined according to the current transmission rate and scheduling of NORM_CMD(CC) messages. The maximum length of the list corresponds to the sender's "NormSegmentSize" parameter for the session. The inclusion of additional entries in the list based on receiver feedback are prioritized with following rules:

1) Receivers that have not yet been provided RTT feedback get first priority. Of these, those with the greatest loss fraction receive precedence for list inclusion.

2) Secondly, receivers that have previously been provided RTT are included with receivers yielding the lowest calculated congestion rate getting precedence.

There are also "cc_flag" values in addition to NORM_CC_FLAG_CLR that are used for other congestion control functions. The NORM_CC_FLAG_CLR flag value is used to mark additional receivers from which the sender would like to have immediate, non-suppressed feedback. These may be receivers which the sender algorithmically identified as potential, future CLRs or which have been pre-configured as potential congestion control points in the network. The NORM_CC_FLAG_RTT indicates the validity of the "cc_rtt" field for the associated receiver node. Normally, this flag will be set since the receivers in the list will typically be receivers from which the sender has received feedback. However, in the case that the NORM sender has been pre-configured with a set of PLR nodes, feedback from those receivers may not yet have been collected and thus the "cc_rtt" and "cc_rate" fields do not contain valid values.

5.5.2.2 NORM_CMD(CC) Feedback Response

Receivers explicitly respond to NORM_CMD(CC) messages in the form of a NORM_ACK(RTT) message. Receivers that are are marked as CLR or PLR nodes in the NORM_CMD(CC) "cc_node_list" immeditately provide feedback in the form of a NORM_ACK to this message. When a NORM_CMD(CC) is received, non-CLR or non-PLR nodes initiate random feedback backoff timeouts similar to that used when the receiver initiates a repair cycle (see Section 5.3) in response to detection of data loss. The goal of the congestion control feedback is to determine the receivers with the lowest congestion control rates. As described in [21], the receiver congestion control feedback (ACK) timeouts can be biased in favor of lower rate receivers (while maintaining effective feedback suppression). Such biasing is not necessarily possible with suppression of NORM_NACK messages since previous data and repair loss history may not be correlated with the current data loss.

The backoff timeout for the congestion control response is picked and biased as follows:

T_backoff = y*r*(K*GRTTsender) + (1 - y)*RandomBackoff(K*GRTTsender, GSIZEsender)

where

"y" is the fraction of (K*GRTT) used to offset the backoff with respect to the sender's current transmission rate. A value of y = 0.25 is recommended.

"r" is adjusted ratio of the local receiver's calculated rate to the sender's current rate. During steady-state congestion control operation, "r" is determined as:

r = (MAX(MIN((Rcalc / Rsender), 0.9), 0.5) - 0.5) / 0.4

During the "slow start" phase of congestion control operation, "r" is determined simply as:

r = Rrecv / Rsender

where "Rrecv" is the measured received rate. The receiver places a value equal to two times this "Rrecv" rate in the "cc_rate" field of its NORM_NACK or NORM_ACK feedback messages during the "slow start" phase of congestion control operation. If the sender chooses this rate as its congestion control rate, this prevents the sender from overshooting an appropriate rate by more than a factor of two during this "slow start" period when receivers have experienced no loss.

The RandomBackoff() algorithm provides a truncated exponentially distributed random number and is described in the NORM Building Block document [11]. The same backoff factor "K" used with the GRTT as for NORM_NACK suppression. As previously noted, a value of K = 4 is generally recommended for ASM operation and K = 6 for SSM operation. A receiver SHALL cancel the backoff timeout and thus its pending transmission of a NORM_ACK(RTT) message under the following conditions:

1) The receiver provides another feedback message (NORM_NACK or NORM_ACK) before the congestion control feedback timeout expires,

2) A "suppressing" NORM_ACK(RTT) message is heard from another receiver or via a NORM_CMD(REPAIR_ADV) message from the sender. The local receiver's feedback is canceled if the rate of the competing feedback (Rfb) is sufficiently close to or less than the local receiver's calculated rate (Rcalc). The local receiver's feedback is canceled when:

   Rcalc > (0.9 * Rfb)

According to [21], this bias of suppression is recommended to help ensure that the receiver with the lowest rate reports, while still maintaining a low volume of feedback from the receiver set.

When the backoff timer expires, the receiver generates a NORM_ACK(RTT) message to provide feedback to the sender and group. This message may be multicast to the group for most effective suppression in ASM topologies or unicast to the sender depending upon how the NORM protocol is deployed and configured. In the congestion control feedback fields of any NORM_ACK or NORM_NACK messages, receivers will include an adjusted version of the sender timestamp from the most recently received NORM_CMD(CC) message and the greatest "cc_sequence" received. The receiver SHALL also set any applicable "cc_flags", its current "cc_rate", and its "cc_rtt" if known. The sender can use the receiver-provided previous "cc_rtt" value to smooth its RTT estimate when it is valid. As noted in [18], a smoothing constant of 0.5 is recommended for regular receivers and 0.9 for CLR (and PLR) receivers from which more rapid feedback is received.

During "slow start" (when the receiver has not yet detected loss from the sender), the receiver uses a value equal to two times its measured rate from the sender in the "cc_rate" field. For steady-state congestion control operation, the receiver "cc_rate" value is based on the equation based value using its current loss event estimate and sender<->receiver RTT information.

After a congestion control feedback message is generated or when the feedback is suppressed, the receiver begins a "holdoff" timeout period during which it will restrain itself from initiating another feedback cycle, even if NORM_CMD(CC) messages are received from the sender (unless the receive becomes marked as a CLR or PLR node). The value of this holdoff timeout period is:

```
                        T_holdoff = (K*GRTT)
```

Thus, non-CLR receivers are constrained to providing explicit congestion control feedback once per K*GRTT intervals. Note, however, that as the session progresses, different receivers will be responding to different NORM_CMD(CC) messages and there will be relatively continuous feedback of congestion control information while the sender is active.

5.5.2.3 Congestion Control Rate Adjustment

During steady-state operation, the sender will directly adjust its transmission rate to the rate indicated by the feedback from its currently selected CLR according to any limitations described in [18]. As noted there, the estimation of parameters (loss and RTT) for the CLR will generally constrain the rate changes possible within acceptable bounds. For rate increases, the sender SHALL observe a maximum rate of increase of one packet per RTT at all times during steady-state operation. Note that the sender SHALL maintain a smoothed RTT estimate for the CLR upon new feedback from the CLR where:

```
              RTT_clr = 0.9 * RTT_clr + 0.1 * RTT_clrNew
```

"RTT_clrNew" is the new RTT calculated from the timestamp in the feedback message received from the CLR. The RTTclr is initialized to RTT_clrNew on the first feedback message received. Note that the same procedure is observed by the sender for PLR receivers and that if a PLR is "promoted" to PLR status, the smoothed estimate can be continued.

There are some additional periods besides steady-state operation which need to be considered in this protocol operation. These periods aare:

1) during session startup,
2) when no feedback is received from the CLR, and
3) when the sender has a break in data transmission.

During session startup, the congestion control operation SHALL observe a "slow start" procedure to quickly approach its fair bandwidth share. An initial sender startup rate is assumed where:

Rinitial = MIN(NormSegmentSize / GRTT, NormSegmentSize) bytes/second.

The rate is increased only when feedback is received from the receiver set. The "slow start" phase proceeds until any receiver provides feedback indicating that loss has occurred. Rate increase during "slow start" is applied as:

Rnew = Rrecv_min

where "Rrecv_min" is the minimum reported receiver rate in the "cc_rate" field of congestion control feedback messages received from the group. Note that during "slow start", receivers use two times their measured rate from the sender in the "cc_rate" field of their feedback. Rate increase adjustment is limited to once per GRTT during slow start.

If the CLR or any receiver intends to leave the group, it will set the NORM_CC_FLAG_LEAVE in its congestion control feedback message as an indication that the sender should not select it as the CLR. When the CLR changes to a lower rate receiver, the sender should immediately adjust to the new lower rate. The sender is limited to increasing its rate at one additional packet per RTT towards a new, higher CLR rate.

The sender should also track the "age" of the feedback it has received from the CLR by comparing its current "cc_sequence" value (Ssender) to the last "cc_sequence" value received from the CLR (Sclr). As the "age" of the CLR feedback increases with no new feedback, the sender SHALL begin reducing its rate once per CLR RTT as a congestion avoidance measure.

The following algorithm is used to determine the decrease in sender rate (Rsender bytes/sec) as the CLR feedback, unexpectedly, excessively ages:

```
Age = Ssender - Sclr;
rate1 = MAX((Rsender - NormSegmentSize), 0.0);  // bytes per sec
rate2 = Rsender * 0.5
if (Age > 4)
     Rsender = MIN(rate1, rate2);
else if (Age > 2)
     Rsender = MAX(rate1, rate2);
```

This rate reduction occurs limited to the lower bound on NORM transmission rate. After NORM_ROBUST_FACTOR consecutive NORM_CMD(CC) rounds without any feedback from the CLR, the sender SHOULD assume the CLR has left the group and pick the receiver with the next lowest rate as the new CLR. Note this assumes that the sender does not have explicit knowledge that the CLR intentionally left the group. After such a CLR timeout, the sender will be transmitting with a minimal rate and should return to slow start as described here for a break in data transmission.

When the sender has a break in its data transmission, it can continue to probe the group with NORM_CMD(CC) messages to maintain RTT collection from the group. This will enable the sender to quickly determine an appropriate CLR upon data transmission restart. However, the sender should exponentially reduce its target rate to be used for transmission restart as time since the break elapses. The target rate SHOULD be recalculated once per CLR RTT as:

```
                    Rsender = Rsender * 0.5;
```

Upon restart, the sender should set the NORM_FLAG_START flag in its NORM_CMD(CC) messages and the group should observer "slow start" congestion control procedures until any receiver experiences a new loss event.

5.5.3 NORM Positive Acknowledgment Procedure

NORM provides an option for the source application to request positive acknowledgment (ACK) of NORM_CMD(ACK_REQ) messages from members of the group. There are a few types of specific acknowledgement requests that are defined for the NORM protocol and a range of acknowledgment request types which left to be defined by the application. One predefined acknowledgement type is the NORM_ACK(WATERMARK) that is used to determine if receivers have acheived completion of reliable reception up through an identified transmission point with respect to the sender's logical sequence of transmission. The NORM_ACK(WATERMARK) acknowledgement may be used to assist in application flow control when the sender has information on a portion of the receiver set. Another predefined acknowledgement type is NORM_ACK(CC), which is used to explicitly provide congestion control feedback in response to NORM_CMD(CC) messages transmitted by the sender. Note the NORM_ACK(CC) response does NOT follow the positive acknowledgement procedure described here. The NORM_CMD(ACK_REQ) and NORM_ACK messages contain an "ack_type" field to identify the type of acknowledgement requested and provided. A range of "ack_type" values is provided for application-defined use. While the application initiates the acknowledgement request and interprets application-defined "ack_type" values, the acknowledgment request and response is conducted with the following procedure.

The NORM positive acknowledgement procedure uses polling by the sender to query the receiver group for response. Note this polling procedure is not intended to scale to very large receiver groups, but could be used in large group setting to query a critical subset of the group. The NORM_CMD(ACK_REQ) message is used for polling and contains a list of NormNodeIds for receivers that should respond to the command. The list of receivers providing acknowledgement is determined by the source application with "a priori" knowledge of participating nodes or via some other application-level mechanism.

The ACK process is initiated by the sender who generates NORM_CMD(ACK_REQ) messages in periodic "rounds". For NORM_ACK(WATERMARK), these requests contain the "object_transport_id", "fec_block_number", and "fec_symbol_id" denoting the watermark transmission point. For application-defined requests, the "ack_req_content" field of the NORM_CMD(ACK_REQ) is set and interpreted by the sender and receiver applications, respectively. In response to the NORM_CMD(ACK_REQ), the listed receivers randomly spread NORM_ACK messages uniformly in time over a window of (1*GRTT). These NORM_ACK messages are typically unicast to the sender.

The ACK process is self-limiting and avoids ACK implosion in that:

1) Only a single NORM_CMD(ACK_REQ) message is generated once per (2*GRTT), and

2) The size of the "acking_node_list" of NormNodeIds from which ACK is requested is limited to a maximum of the sender NormSegmentSize setting per round of the positive acknowledgement process.

Because the size of the included list is limited to the sender's NormSegmentSize setting, multiple NORM_CMD(ACK_REQ) rounds may be required to achieve responses from all receivers specified. The content of the attached NormNodeId list will be dynamically updated as this process progresses and ACKs are received from the specified receiver set. Thus, as the sender receives responses from receivers, it eliminates them from the subsequent NORM_CMD(ACK_REQ) message payload list and adds in any pending receiver NormNodeIds keeping within the NormSegmentSize limitation of the list size. Each receiver is queried a maximum number of times (NORM_ROBUST_FACTOR, by default). Receivers not responding within this number of repeated requests are removed from the payload list to make potential room for other receivers pending acknowledgement. The transmission of the NORM_CMD(ACK_REQ) is repeated until no further responses are required or until the repeat threshold is exceeded for all pending receivers. The transmission of NORM_CMD(ACK_REQ) messages to conduct the positive acknowledgment process is multiplexed with ongoing sender data transmissions. However, the positive acknowledgment process may be interrupted in response to negative acknowledgement repair requests (NACKs) received from receivers during the acknowledgment period. The ACK process is resumed once any pending repairs have been transmitted.

In the case of NORM_CMD(ACK_REQ(WATERMARK)) commands, receivers will not ACK until they have received complete transmission of all data up to and including the watermark transmission point. All receivers SHALL interpret the watermark point provided in the request in the same manner as the transmission point given in NORM_CMD(FLUSH) messages and NACK for repairs if needed.

5.5.4 Group Size Estimation

NORM sender messages contain a "gsize" field that is a representation of the group size and is used in scaling random backoff timer ranges. The use of the group size estimate within the NORM protocol does not require a precise estimation and works reasonably well if the estimate is within an order of magnitude of the actual group size. By default, the NORM sender group size estimate may be administratively configured. Also given the expected scalability of the NORM protocol for general use, a default value of 10,000 is recommended for use as the group size estimate.

It is possible that group size may be algorithmically approximated from the volume of congestion control feedback messages which follow the exponentially weighted random backoff. However, the specification of such an algorithm is currently beyond the scope of this document.

5.5.5 Operation with Generic Router Assist (GRA)

NORM packet formats will be extended to allow for operation with GRA reliable multicast functions. Additional NACK suppression and selective sub-casting of repair transmissions in the network will be possible with GRA. (Section 5.4.2 discusses some NORM mechanisms related to this). Additional details will be provide in future versions of this document as GRA specifications mature.

6.0 Security Considerations

The same security considerations that apply to the NORM, FEC, and TFMCC building blocks also apply to the NORM protocol. In addition to vulnerabilities that any IP and IP multicast protocol implementation may be generally subject to, the NACK based feedback of NORM may be exploited by replay attacks which force the NORM sender to unnecessarily transmit repair information. This MAY be addressed by network layer IP security implementations that guard against this potential security exploitation. It is RECOMMENDED that such IP security mechanisms be used when available. Another possible approach is for NORM senders to use the "sequence" field from the NORM Common Message Header to detect replay attacks. This can be accomplished if the sender is willing to maintain state on receivers which are NACKing. A cache of receiver state may provide some protection against replay attacks. Note that the "sequence" field should be incremented by NormNodes with independent values for "sender" messages versus "receiver" messages so that the congestion control loss estimation function of the "sequence" field can be preserved for sender messages when receiver messages are unicast to the sender.

While NORM does leverage FEC-based repair for scalability, this does not alone guarantee integrity of received data. Application-level integrity-checking of data content is highly RECOMMENDED.

The NORM protocol is compatible with the use of the IP security (IPSEC) architecture described in [23].

7.0 Suggested Use

The present NORM protocol is seen as useful tool for the reliable data transfer over generic IP multicast services. It is not the intention of the authors to suggest it is suitable for supporting all envisioned multicast reliability requirements. NORM provides a simple and flexible framework for multicast applications with a degree of concern for network traffic implosion and protocol overhead efficiency. NORM-like protocols have been successfully demonstrated within the MBone for bulk data dissemination applications, including weather satellite compressed imagery updates servicing a large group of receivers and a generic web content reliable "push" application.

In addition, this framework approach has some design features making it attractive for bulk transfer in asymmetric and wireless internetwork applications. NORM is capable of successfully operating independent of network structure and in environments with high packet loss, delay, and misordering. Hybrid proactive/reactive FEC-based repairing improve protocol performance in some multicast scenarios. A sender-only repair approach often makes additional engineering sense in asymmetric networks. NORM's unicast feedback capability may be suitable for use in asymmetric networks or in networks where only unidirectional multicast routing/delivery service exists. Asymmetric architectures supporting multicast delivery are likely to make up an important portion of the future Internet structure (e.g., DBS/cable/PSTN hybrids) and efficient, reliable bulk data transfer will be an important capability for servicing large groups of subscribed receivers.

8.0 References

[1] Kermode, R., Vicisano, L., "Author Guidelines for Reliable Multicast Transport (RMT) Building Blocks and Protocol Instantiation documents", RFC 3269, April 2002.

[2] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[3] Mankin, A., Romanow, A., Bradner, S. and V. Paxson, "IETF Criteria for Evaluating Reliable Multicast Transport and Application Protocols", RFC 2357, June 1998.

[4] Whetten, B., Vicisano, L., Kermode, R., Handley, M., Floyd S. and Luby, M., "Reliable Multicast Transport Building Blocks for One-to-Many Bulk-Data Transfer", RFC 3048, January 2001.

[5] Handley, M. and V. Jacobson, "SDP: Session Description Protocol", RFC 2327, April 1998.

[6] Handley, M., Perkins, C. and E. Whelan, "Session

Announcement Protocol", RFC 2974, October 2000.

[7] S. Pingali, D. Towsley, J. Kurose, "A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols", In Proc. INFOCOM, San Francisco CA, October 1993.

[8] Luby, M., Vicisano, L., Gemmell, J., Rizzo, L., Handley, M. and J. Crowcroft, "The Use of Forward Error Correction (FEC) in Reliable Multicast", RFC 3453, December 2002.

[9] J. Macker, R. Adamson, "The Multicast Dissemination Protocol (MDP) Toolkit", Proc. IEEE MILCOM 99, October 1999.

[10] J. Nonnenmacher and E. Biersack, "Optimal Multicast Feedback", Proc. IEEE INFOCOMM, p. 964, March/April 1998.

[11] J. Macker, R. Adamson, "Quantitative Prediction of Nack Oriented Reliable Multicast (NORM) Feedback", Proc. IEEE MILCOM 2002, October 2002.

[12] T. Speakman, L. Vicisano, "Reliable Multicast Transport Building Block Generic Roouter Assist - Signalling Protocol Specification", Internet Draft draft-ietf-rmt-bb-gra-signalling-01.txt, January 2003, work in progress. Citation for informational purposes only.

[13] Deering, S., "Host Extensions for IP Multicasting", STD 5, RFC 1112, August 1989.

[14] Holbrook, H. W., "A Channel Model for Multicast", Ph.D. Dissertation, Stanford University, Department of Computer Science, Stanford, California, August 2001.

[15] B. Adamson, C. Bormann, M. Handley, and J. Macker, "NACK-Oriented Reliable Multicast (NORM) Protocol Building Blocks", Internet Draft draft-ietf-rmt-bb-norm-05.txt, March 2003, work in progress. Citation for informational purposes only.

[16] M. Luby, L. Vicisano, J. Gemmell, L. Rizzo, M. Handley, and J. Crowcroft, "The Use of Forward Error Correction (FEC) in Reliable Multicast", RFC 3453, December 2002.

[17] M. Luby, L. Vicisano, J. Gemmell, L. Rizzo, M. Handley, and J. Crowcroft, "Forward Error Correction (FEC) Building BLock", RFC 3452, December 2002.

[18] J. Widmer, M. Handley, "TCP-Friendly Multicast Congestion Control (TFMCC) Protocol Specification", Internet Draft draft-ietf-rmt-bb-tfmcc-01.txt, November 2002, work in progress. Citation for informational purposes only.

[19] D. Gossink, J. Macker, "Reliable Multicast and Integrated Parity Retransmission with Channel Estimation", IEEE GLOBECOMM 98', September 1998.

[20] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", RFC 1889, January 1996.

[21] J. Widmer and M. Handley, "Extending Equation-Based Congestion Control to Multicast Applications", Proc ACM SIGCOMM 2001, San Diego, August 2001.

[22] L. Rizzo, "pgmcc: A TCP-Friendly Single-Rate Multicast Congestion Control Scheme", Proc ACM SIGCOMM 2000, Stockholm, August 2000.

[23] S. Kent and R. Atkinson, "Security Architecture for the Internet Protocol", RFC 2401, November 1998.

7.0 Authors' Addresses

Brian Adamson
adamson@itd.nrl.navy.mil
Naval Research Laboratory
Washington, DC, USA, 20375

Carsten Bormann
cabo@tellique.de
Tellique Kommunikationstechnik GmbH
Gustav-Meyer-Allee 25 Geb ude 12
D-13355 Berlin, Germany

Mark Handley
mjh@aciri.org
1947 Center Street, Suite 600
Berkeley, CA 94704

Joe Macker
macker@itd.nrl.navy.mil
Naval Research Laboratory
Washington, DC, USA, 20375

A.2 The NORM Protocol's GRTT Measurement Procedure

3.7.1 One-to-Many Sender GRTT Measurement

The goal of this form of RTT measurement is for the sender to learn the GRTT among the receivers who are actively participating in NORM operation. The set of receivers participating in this process may be the entire group or some subset of the group determined from another mechanism within the protocol instantiation. An approach to collect this GRTT information follows.

The sender periodically polls the group with a message (independent or "piggy-backed" with other transmissions) containing a <sendTime>

timestamp relative to an internal clock at the sender. Upon reception of this message, the receivers will record this <sendTime> timestamp and the time (referenced to their own clocks) at which it was received <recvTime>. When the receiver provides feedback to the sender (either explicitly or as part of other feedback messages depending upon protocol instantiaon specification), it will construct a "response" using the formula:

```
          grttResponse = sendTime + (currentTime - recvTime)
```

where the <sendTime> is the timestamp from the last probe message received from the source and the (currentTime - <recvTime) is the amount of time differential since that request was received until the receiver generated the response.

The sender processes each receiver response by calculating a current RTT measurement for the receiver from whom the response was received using the following formula:

```
               receiverRtt = currentTime - grttResponse
```

During the each periodic GRTT probing interval, the source keeps the peak round trip estimate from the set of responses it has received. The GRTT estimate should be filtered to be conservative towards maintaining an estimate biased towards the greatest receiver RTT measurements received. A conservative estimate of GRTT maximizes the efficiency redundant NACK suppression and repair aggregation. The update to the source's ongoing estimate of GRTT is done observing the following rules:

1) If a receiver's response round trip calculation is greater than the current GRTT estimate AND current peak, the response value is immediately fed into the GRTT update filter given below. In any case, the source records the "peak" receiver RTT measurement for the current probe interval.

2) At the end of the response collection period (i.e. the GRTT probe interval), if the recorded "peak" response is less than the current GRTT estimate AND this is the third consecutive collection period with a peak less than the current GRTT estimate the recorded peak is fed into the GRTT update. (Implicitly, Rule #1 was applied otherwise so no new update is required).

3) At the end of the response collection period, the peak tracking value is set to either ZERO if the "peak" is greater than or equal to the current GRTT estimate (i.e. Already incorporated into the filter under Rule #1) or kept the same if its value is less than the current GRTT estimate AND was not yet incorporated into the GRTT update filter according to Rule #2. Thus for decreases in the source's estimate of GRTT, the "peak" is tracked across three consecutive probe intervals.

The following GRTT update filter is used to incorporate new peak responses into the the GRTT estimate:

```
        if (peak > current_estimate)
            current_estimate = 0.25 * current_estimate + 0.75 * peak;
        else
            current_estimate = 0.75 * current_estimate + 0.25 * peak;
```

This update method is biased towards maintaining an estimate of the worst-case round trip delay. The reason the GRTT estimate is reduced only after 3 consecutive collection periods with smaller response peaks is to be conservative where packet loss may have resulted in lost response messages. And then the reduction is additionally conservatively weighted using the averaging filter from above.

The GRTT collection period (i.e. period of probe transmission) could be fixed at a value on the order of that expected for group membership and/or network topology dynamics. For robustness, more rapid probing could be used at protocol startup before settling to a less frequent, steady-state interval. Optionally, an algorithm may be developed to adjust the GRTT collection period dynamically in response to the current GRTT estimate (or variations in it) and to an estimation of packet loss. The overhead of probing messages could then be reduced when the GRTT estimate is stable and unchanging, but be adjusted to track more dynamically during periods of variation with correspondingly shorter GRTT collection periods.

In summary, although NORM repair cycle timeouts are based on GRTT, it should be noted that convergent operation of the protocol does not _strictly_ depend on highly accurate GRTT estimation. The current mechanism has proved sufficient in simulations and in the environments where NORM-like protocols have been deployed to date. The estimate provided by the algorithm tracks the peak envelope of actual GRTT (including operating system effect as well as network delays) even in relatively high loss connectivity. The steady-state probing/update interval may potentially be varied to accommodate different levels of expected network dynamics in different environments.

Appendix B

The Real-Time Maude Specification of the NORM Protocol

This appendix contains the modules of the Real-Time Maude specification of the NORM multicast protocol. Section B.1 contains the modules that define the tick rule, time advance in an object configuration, the network topology and communication, the sender and receiver messages, and the functions for calculating the receiver feedback suppression timeouts. Sections B.2 and B.3 contain the specifications of the NORM protocol's RTT component and data transmission component, respectively. Finally, the module that combines these specifications into an overall specification of the protocol is included in Section B.4. In order to save space, I have removed some of my original comments in the modules.

B.1 Common Modules

```
(tomod TIMED-OO-SYSTEM is
  inc LTIME-INF .

  sort OidSet .
  subsort Oid < OidSet .
  op none : -> OidSet .
  op __ : OidSet OidSet -> OidSet [assoc comm id: none] .

  op delta : Configuration Time -> Configuration [frozen (1)] .
  op mte : Configuration -> TimeInf [frozen (1)] .

  vars NECF NECF' : NEConfiguration .
  var T : Time .
  eq delta(none, T) = none .
  eq delta(NECF NECF', T) = delta(NECF, T) delta(NECF', T) .
  eq mte(none) = INF .
  eq mte(NECF NECF') = min(mte(NECF), mte(NECF')) .

endtom)
```

```
(tomod DETERMINISTIC-TICK-RULE is
  inc TIMED-OO-SYSTEM .

  var OC : NEObjectConfiguration .

  crl [tick] :
    {OC} => {delta(OC, mte(OC))} in time mte(OC) if mte(OC) =/= INF .

endtom)


(fmod TIMEOUT-FUNCTIONS is
  inc NAT . inc RAT . inc CONVERSION .

  vars N N' N'' N''' : Nat . vars NZN NZN' : NzNat . vars  R R' : Rat .


  *** KNUTH'S RANDOM FUNCTION (FROM OELVECZKYS'S THESIS) ***

  op random : Nat -> Nat .
  eq random(N) = ((104 * N) + 7921) rem 10609 .


  *** LOG AND EXP FOR RATIONAL NUMBERS ***

  op log : Rat -> Rat .
  op exp : Rat -> Rat .

  eq log(R) = rat(log(float(R))) .
  eq exp(R) = rat(exp(float(R))) .


  *** From rational numbers to natural numbers.
  op nat : Rat -> Nat .
  eq nat(R) = trunc(R + 1/2) .


  *** RANDOM BACKOFF ALGORITHM ***

  *** The randomBackoff operator is used by the receivers to set their
  *** backoff timers in the data transmission component.

  *** Usage: maxBackoff(NormRobustFactor, GRTT)
  op maxBackoff : NzNat Nat -> Nat .
  eq maxBackoff(NZN, N) = NZN * N .


  op lambda : Rat -> Rat .
  eq lambda(R) = log(R) + 1 .


  *** Usage: randomFrac(seed)
  *** Returns a fraction value from 0 to 1.
  op randomFrac : Nat -> Rat .
  eq randomFrac(N) = random(N) rem (MAX-VALUE + 1) / MAX-VALUE .
  op MAX-VALUE : -> Nat .
  eq MAX-VALUE = 14 .


  *** Find a random value between 0-1:
  *** Usage: randomUpTo(seed, upper)
  op randomUpTo : Nat Rat -> Rat .
  eq randomUpTo(N, R) = randomFrac(N) * R .


  *** Usage: x(seed, NormRobustFactor, GRTT, GSIZE)
  op x : Nat NzNat Nat NzNat -> Rat .
  eq x(N, NZN, N', NZN') =
    randomUpTo(N, lambda(NZN') / maxBackoff(NZN, N'))
    + lambda(NZN') / (maxBackoff(NZN, N') * (exp(lambda(NZN')) - 1)) .


  *** Usage: randomBackoff(seed, NormRobustFactor, GRTT, GSIZE)
  op randomBackoff : Nat NzNat Nat NzNat -> Nat .
  eq randomBackoff(N, NZN, N', NZN') =
    nat((maxBackoff(NZN, N') / lambda(NZN'))
         * log(x(N, NZN, N', NZN')
               * (exp(lambda(NZN')) - 1)
               * (maxBackoff(NZN, N') / lambda(NZN')))) .



  *** RTT FEEDBACK FUNCTION ***

  *** The RTTbackoff operator is used by the receivers to set their
  *** backoff timers in the RTT component.

  *** min and max for rational numbers
  op minimum : Rat Rat -> Rat [assoc comm] .
  op maximum : Rat Rat -> Rat [assoc comm] .
  ceq minimum(R, R') = R if R <= R' .
  ceq maximum(R, R') = R if R >= R' .

  op r : Rat Rat -> Rat .
  eq r(R, R') = (maximum(minimum((R / R'), 9/10), 1/2) - 1/2) / 2/5 .


  *** Usage: RTTbackoff(seed, NormRobustFactor, GRTT,
  ***                   GSIZE, rcvRateInKbps, sndRateInKbps)
  op RTTbackoff : Nat NzNat Nat NzNat Rat Rat -> NzNat .
  eq RTTbackoff(N, NZN, N', NZN', R, R') =
    nat(1/4 * r(R, R') * maxBackoff(NZN, N')
        + (1 - 1/4) * randomBackoff(N, NZN, N', NZN')) .

endfm)


(tomod MESSAGES is
  inc TIMED-OO-SYSTEM .             inc NAT-TIME-DOMAIN-WITH-INF .

  vars T T' : Time .                var M : Msg .
  vars ML ML' : MsgList .           vars DUI DUI' : DataUnitId .
```

```
var DUIL DUIL' : DataUnitIdList . vars NZN NZN' NZN'' NZN''' : NzNat .

*** Objects store their messages in lists. New messages are entered
*** from the right, and the leftmost message is the oldest one in the list.
sort MsgList .
subsort Msg < MsgList .
op nil : -> MsgList [ctor] .
op _++_ : MsgList MsgList -> MsgList [ctor assoc id: nil] .


*** The effect of time on a message list
op delta : MsgList Time -> MsgList [frozen (1)] .
op delta : Msg Time -> Msg [frozen (1)] .
eq delta((nil).MsgList, T) = (nil).MsgList .
ceq delta(ML ++ ML', T) =
   delta(ML, T) ++ delta(ML', T) if ML =/= nil /\ ML' =/= nil .
eq delta(dly(M, T), T') = dly(M, T monus T') .


*** A message passing through a link or stored in the buffer of a
*** router has a delay value attached to it.
msg dly : Msg Time -> Msg .


*** Least delay of a message in a message list.
op leastDly : MsgList -> TimeInf .
eq leastDly(nil) = INF .
*** the leftmost (oldest) msg has the least delay:
eq leastDly(dly(M, T) ++ ML) = T .


*** Greatest delay of a message in a message list.
op greatestDly : MsgList -> TimeInf .
eq greatestDly(nil) = 0 .
*** the rigthmost (newest) msg has the greatest delay:
eq greatestDly(ML ++ dly(M, T)) = T .


*** IDENTIFICATION OF DATA CONTENT

*** A data segment is identified with a pair
*** (objectIdentifier :: segmentIdentifier).

sort DataUnitId .
op _::_ : Nat Nat -> DataUnitId .
sort DataUnitIdList . subsort DataUnitId < DataUnitIdList .
op nil : -> DataUnitIdList [ctor] .
op _;_ : DataUnitIdList DataUnitIdList
         -> DataUnitIdList [ctor assoc id: nil] .


*** MESSAGE SIZES

sorts ControlPacket DataPacket .
subsort ControlPacket < Msg . subsort DataPacket < Msg .
```

```
  *** NORM SENDER MESSAGES

  *** Usage: DATA(dataUnitId, noOfSegmentsInObject, grtt, repairFlag)
  msg DATA : DataUnitId NzNat Time Bool -> DataPacket .

  *** Usage: FLUSH(dataUnitId, grtt, eotFlag)
  msg FLUSH : DataUnitId Time Bool -> ControlPacket .

  *** Usage: SQUELCH(earliestDUIvalidForRepair, grtt)
  msg SQUELCH : DataUnitId Time -> ControlPacket .

  *** Usage: CC(sendTime/timestamp, grtt, sendRateInKbps)
  msg CC : Time Time Nat -> ControlPacket .


  *** NORM RECEIVER MESSAGES

  *** Usage: NACK(dataUnitIdList, adjustedTimestamp, CLRflag)
  msg NACK : DataUnitIdList Time Bool -> ControlPacket .

  *** Usage: ACK(adjustedTimestamp, recRateInKbps, CLRflag)
  msg ACK : Time Nat Bool -> ControlPacket .

endtom)


(tomod NODES is
  inc TIMED-OO-SYSTEM .        inc NAT-TIME-DOMAIN-WITH-INF .
  inc TIMEOUT-FUNCTIONS .

  *** General properties of nodes.
  class RootOrLeaf | clock : Time, NormRobustFactor : NzNat,
                     GRTT : Time, gsize : NzNat .
  class Parent | children : OidSet  .
  class Child | parent : Oid .
  class RandomSeed | randomSeed : Nat .
  class Sender | sendRate : Time .
  class Receiver | CLR : Bool . *** The Current Limiting Receiver

  subclass Sender < Parent RootOrLeaf .
  subclass Receiver < RootOrLeaf Child RandomSeed .

endtom)


(tomod NETWORK is
  inc TIMED-OO-SYSTEM .      inc NAT-TIME-DOMAIN-WITH-INF .
  inc NODES .                inc MESSAGES .

  vars O O' R L : Oid .      var OS : OidSet .
  var M : Msg .              vars ML ML' : MsgList .
  var CP : ControlPacket .   var DP : DataPacket .
  vars N N' : Nat .          var NZN : NzNat .
  vars T T' : Time .         var NZT : NzTime .


  *** TRANSMITTING ACROSS A LINK
  *** A packet is put in a "wrapper" when it is released into the configuration.
  msg intoLink_from_to_ : Msg Oid Oid -> Msg . ***message sent to link
  msg outOfLink_from_to_ : Msg Oid Oid -> Msg . ***message received from link


  *** Routers and senders use a multisend wrap to broadcast a packet
  *** to a subset of the multicast group (i.e., their connections).
  msg multisend_from_to_ : Msg Oid OidSet -> Configuration .
  eq multisend M from O to none = none .
  eq multisend M from O to (O' OS) =
     (intoLink M from O to O') (multisend M from O to (OS ex O')) .


  *** Pick out all occurences of an Oid from a set of Oids.
  op _ex_ : OidSet Oid -> OidSet .
  eq none ex O = none .
  eq (O OS) ex O' =
    if O == O' then (OS ex O')
    else O (OS ex O') fi .
  eq O O = O .


  *** LINKS

  class Link |
    upNode : Oid, downNode : Oid,
    upstream : MsgList,
    downstream : MsgList,
    propDelay : NzTime,
    bandwidth : NzNat . *** in Megabits pr. sec.


  *** Transmission delay of a packet in a link is the packet size
  *** divided by the bandwidth. Data packets are 1500 bytes,
  *** control packets are 64 bytes large.
  op transDelay : Msg NzNat -> Time .

  *** Packet size of 64 bytes (control packet):
  eq transDelay(CP, NZN) = ((64 * 8) + ((NZN * 1000) monus 1)) quo (NZN * 1000) .
  *** Packet size of 1500 bytes (data packet):
  eq transDelay(DP, NZN) = ((1500 * 8) + ((NZN * 1000) monus 1)) quo (NZN * 1000) .


  *** The next two rules model how messages are entered into a link.

  crl [msgFromUpNode] :
    (intoLink M from O to O')
    < L : Link | upNode : O, downNode : O', downstream : ML,
                 propDelay : NZT, bandwidth : NZN >
    =>
    < L : Link |
    downstream : ML ++ dly(M, max(NZT, greatestDly(ML)) + transDelay(M, NZN)) >
    if leastDly(ML) =/= 0 .


  crl [msgFromDownNode] :
   (intoLink M from O to O')
    < L : Link | downNode : O, upNode : O', upstream : ML,
                 propDelay : NZT, bandwidth : NZN >
    =>
    < L : Link |
    upstream : ML ++ dly(M, max(NZT, greatestDly(ML)) + transDelay(M, NZN)) >
    if leastDly(ML) =/= 0 .


  *** The next two rules model how messages are delivered from a link.

  rl [msgToUpNode] :
    < L : Link | upNode : O, downNode : O', upstream : dly(M, 0) ++ ML >
    =>
    < L : Link | upstream : ML >
    (outOfLink M from O' to O) .


  rl [msgToDownNode] :
    < L : Link | downNode : O, upNode : O', downstream : dly(M, 0) ++ ML >
    =>
    < L : Link | downstream : ML >
    (outOfLink M from O' to O) .


  *** Timed behaviour of a link
  eq delta(< L : Link | downstream : ML, upstream : ML' >, T) =
           < L : Link | downstream : delta(ML, T), upstream : delta(ML', T) > .
  eq mte(< L : Link | downstream : ML, upstream : ML' >) =
           min(leastDly(ML), leastDly(ML')) .


  *** ROUTERS

  class Router |
    buffer : MsgList, bufferCap : Nat, queuingDelay : Time .

  *** Routers have a parent and a set of children.
  subclass Router < Parent  Child .

  *** Incoming packets are stored with info about who sent it to the router.
  msg forward : Oid Msg -> Msg . *** forward Msg from Oid.


  *** A message is stored in the routers buffer if its capacity is
  *** more than zero. Otherwise, the message is discarded.

  rl [bufferOrDrop] :
    (outOfLink M from O to R)
    < R : Router | buffer : ML, bufferCap : N, queuingDelay : N' >
    =>
    if N > 0
     then < R : Router | buffer : ML ++ dly(forward(O, M), N'),
                         bufferCap : sd(N, 1) >
     else < R : Router | > fi .
  *** A message is forwarded by the router.

  rl [forward] :
  < R : Router | parent : O, children : OS,
                buffer : dly(forward(O', M), 0) ++ ML, bufferCap : N >
  =>
  < R : Router | buffer : ML, bufferCap : N + 1 >
  if O' == O
   then (multisend M from R to OS)
   else (multisend M from R to O (OS ex O')) fi .


  *** Timed behaviour of routers
  eq delta(< R : Router | buffer : ML >, T) =
           < R : Router | buffer : delta(ML, T) > .
  eq mte(< R : Router | buffer : ML >) = leastDly(ML) .

endtom)
```

B.2 The RTT Specification

```
(tomod NORM-RTT is
  inc NAT-TIME-DOMAIN-WITH-INF .     inc TIMED-OO-SYSTEM .
  inc DETERMINISTIC-TICK-RULE .      inc MESSAGES .
  inc NODES .                        inc NETWORK .
  inc TIMEOUT-FUNCTIONS .

  vars S R O O' : Oid .              var OS : OidSet .
  vars N N' N'' N''' N'''' : Nat .   vars NZN NZN' : NzNat .
  vars T T' T'' T''' T'''' : Time .  var NZT : NzTime .
  vars TI TI' : TimeInf .            var DUIL : DataUnitIdList .
  var M : Msg .                      vars ML ML' : MsgList .
  vars B B' : Bool .


  *** THE SENDER ***

  class RTTsender |
    sendRateInKbps : Nat, *** transmission rate in kbps
    CCtransTimer : TimeInf,
    peakRTT : Time,
    CLRresponse : Bool,
    lowPeakRTTcounter : Nat .

  subclass RTTsender < Sender .

  class RTTsenderAlone .  subclass RTTsenderAlone < RTTsender .


  *** INITIALIZE RTT COLLECTION

  *** The sender initializes RTT collection. The first CC messages are
  *** sent every 500 ms until some receiver provides feedback.

  crl [initializeRTTcollection] :
    < S : RTTsender |
      children : OS, clock : T, sendRateInKbps : N, GRTT : 500,
      sendRate : T', CCtransTimer : TI, peakRTT : 0 >
    =>
    < S : RTTsender |
      CCtransTimer : 500 >
    (multisend CC(T, max(500, T'), N) from S to OS)
    if TI == INF or TI == 0 .


  *** UPDATING GRTT AT THE END OF THE RTT COLLECTION PERIOD

  *** At the end of a RTT collection period, the sender decides whether to
  *** update the GRTT estimate, and transmits a new CC message.

  crl [endOfRTTcollectionPeriod] :
    < S : RTTsender |
      children : OS, clock : T, GRTT : T'',
      sendRate : T''', sendRateInKbps : N', CCtransTimer : 0,
      peakRTT : T', CLRresponse : B, lowPeakRTTcounter : N >
    =>
    (if B == true and T''' <= T'
    then
    (< S : RTTsender |
       CCtransTimer : T',
       peakRTT : (if (T' >= T'') then 0 else T' fi),
       lowPeakRTTcounter : (if ((T' >= T'') or (N == 2))
                           then 0 else (N + 1) fi),
       GRTT : (if ((T' < T'') and (N == 2)) then updateGRTT(T'', T')
              else T'' fi) >
    *** The CC contains the most recent GRTT estimate
 (multisend CC(T, max((if ((T' < T'') and (N == 2)) then updateGRTT(T'', T')
               else T'' fi) , T'''), N') from S to OS))
     else
    (< S : RTTsender |
       CCtransTimer : max((if ((T' < T'') and (N == 2)) then
         updateGRTT(T'', T') else T'' fi), T'''),
       peakRTT : (if (T' >= T'') then 0 else T' fi),
       lowPeakRTTcounter : (if ((T' >= T'') or (N == 2))
                           then 0 else (N + 1) fi),
       GRTT : (if ((T' < T'') and (N == 2)) then updateGRTT(T'', T')
              else T'' fi) >
 (multisend CC(T,  max((if ((T' < T'') and (N == 2)) then updateGRTT(T'', T')
               else T'' fi), T'''), N') from S to OS)) fi)
    if T' =/= 0 or (T' == 0 and T'' < 500) .


  *** UPDATING GRTT DURING THE RTT COLLECTION PERIOD

  *** If the sender receives a RTT value which is larger than the GRTT estimate
  *** and the current peak RTT, the GRTT is updated immediately. Otherwise,
  *** the RTT is simply stored.

  var RECEIVED-RTT : Time .

  *** Receives explicit feedback, i.e., an ACK message.
  crl [receiveAdjustedTimestamp1] :
    (outOfLink ACK(T, N, B) from O to S)
    < S : RTTsender |
      clock : T', GRTT : T''', peakRTT : T'' >
    =>
    (if RECEIVED-RTT > T''' and RECEIVED-RTT > T'' ***currTime - grttResponse
    then
     *** update GRTT
    < S : RTTsender |
      peakRTT : RECEIVED-RTT, CLRresponse : B,
      GRTT : updateGRTT(T''', RECEIVED-RTT) >
    else
     *** keep new peak if larger than old peak
    < S : RTTsender |
      peakRTT : (if RECEIVED-RTT > T'' then RECEIVED-RTT else T'' fi),
      CLRresponse : B > fi)
  if RECEIVED-RTT := (T' monus T) .


  *** Receives implicit feedback, i.e., a NACK message.
  crl [receiveAdjustedTimestamp2] :
    (outOfLink NACK(DUIL, T, B) from O to S)
    < S : RTTsenderAlone |
      clock : T', GRTT : T''', peakRTT : T'' >
    =>
    (if RECEIVED-RTT > T''' and RECEIVED-RTT > T''
    then
     *** update GRTT
    < S : RTTsenderAlone |
      peakRTT : RECEIVED-RTT, CLRresponse : B,
      GRTT : updateGRTT(T''', RECEIVED-RTT) >
    else
     *** keep new peak if larger than old peak
    < S : RTTsenderAlone |
      peakRTT : (if RECEIVED-RTT > T'' then RECEIVED-RTT else T'' fi),
      CLRresponse : B > fi)
  if RECEIVED-RTT := (T' monus T) .


  *** CALCULATE NEW GRTT

  *** Usage: updateGRTT(GRTTestimate, peakRTT) -> newGrtt
  op updateGRTT : Nat Nat -> Time .
  eq updateGRTT(N, N') = nat(if N' > N then 1/4 * N + 3/4 * N'
                             else 3/4 * N + 1/4 * N' fi) .


  *** THE RECEIVER ***

  class RTTreceiver |
    ACKtimer : TimeInf,
    ACKholdoffTimer : TimeInf,
    timestamp : Time,
    receivedTimestamp : Time,
    rcvRateInKbps : Nat, *** reception rate in kbps
    sndRateInKbps : Nat . ***sender's rate

  subclass RTTreceiver < Receiver .
```

```
class RTTreceiverAlone | nacksToBeSent : MsgList .
subclass RTTreceiverAlone < RTTreceiver .

*** Used in stand-alone mode to simulate NACKs:
msg NACK : -> ControlPacket .


*** CURRENT LIMITING RECEIVER FEEDBACK:

*** The current limiting receiver (CLR) provides immediate feedback.

rl [CLRfeedback] :
  (outOfLink CC(T, T', N) from O to R)
  < R : RTTreceiver |
    parent : O, clock : T'', GRTT : T''', CLR : true, rcvRateInKbps : N' >
  =>
  < R : RTTreceiver |
    GRTT : T', sndRateInKbps : N, timestamp : T, receivedTimestamp : T'' >
  (intoLink ACK(T, N', true) from R to O) .


*** NON-CURRENT LIMITING RECEIVER FEEDBACK:

*** STARTS RTT FEEDBACK CYCLE

*** The receiver sets its feedback timer provided it is not running, and
*** provided its holdoff timer is turned off.

rl [initializeACKcycle] :
  (outOfLink CC(T, T'', N) from O to R)
  < R : RTTreceiver |
    clock : T', gsize : NZN, randomSeed : N',
    NormRobustFactor : NZN', rcvRateInKbps : N'',
    ACKtimer : INF, ACKholdoffTimer : INF, CLR : false >
  =>
  < R : RTTreceiver |
    timestamp : T, receivedTimestamp : T', GRTT : T'',
    randomSeed : random(N'), sndRateInKbps : N,
    ACKtimer : RTTbackoff(random(N'), NZN', T'', NZN, N'', N) > .


*** IGNORES CCS WHILE ACK TIMER IS RUNNING

*** If the receiver receives a new CC message while its feedback timer is
*** running, it stores the new GRTT value. This is not stated explicitly
*** in the original specification.

rl [receivesCCduringBackoffTimeout] :
  (outOfLink CC(T, T', N) from O to R)
  < R : RTTreceiver | ACKtimer : NZT >
  =>
  < R : RTTreceiver | GRTT : T' > .


*** CANCELS ACK DURING CYCLE
*** The receiver cancels its pending ACK because it is about to send a
*** NACK message. It includes the RTT feedback in the repair request.

rl [cancelACK1] :
  < R : RTTreceiverAlone |
    parent : O, clock : T, timestamp : T', receivedTimestamp : T'',
    ACKtimer : TI, ACKholdoffTimer : TI', NormRobustFactor : NZN, GRTT : N,
    nacksToBeSent : dly(NACK, 0) ++ ML, CLR : B >
  =>
  < R : RTTreceiverAlone |
    ACKtimer : (if TI =/= INF and TI =/= 0 then INF else TI fi),
    ACKholdoffTimer : (if TI =/= INF and TI =/= 0 then NZN * N else INF fi),
    nacksToBeSent : ML >
  (intoLink NACK(nil, (if T' == 0 then 0 else T' + (T monus T'') fi), B)
  from R to O) .


*** The receiver cancels its pending ACK because another receiver has
*** provided RTT feedback to the sender.

rl [cancelACK2] :
  (outOfLink ACK(T, N, B) from O to R)
  < R : RTTreceiver |
    rcvRateInKbps : N', ACKtimer : NZT, ACKholdoffTimer : INF,
    NormRobustFactor : NZN, GRTT : NZN' >
  =>
  (if N' > (N monus (N quo 10))
  *** cancelled if the competing rate is sufficiently close to or less
  *** than N', i.e. N' > N * 0,9
  then
  < R : RTTreceiver | ACKtimer : INF, ACKholdoffTimer : NZN * NZN' >
  else
  < R : RTTreceiver | > fi) .


*** FEEDBACK TIMER EXPIRES

*** The receiver sends an ACK message with the adjusted timestamp.

rl [sendAjustedTimestamp] :
  < R : RTTreceiver |
    parent : O, clock : T, timestamp : T', receivedTimestamp : T'',
    rcvRateInKbps : N, NormRobustFactor : NZN, GRTT : NZN',
    ACKtimer : 0, ACKholdoffTimer : INF >
  =>
  < R : RTTreceiver |
    ACKtimer : INF, ACKholdoffTimer : NZN * NZN' >
  (intoLink ACK(T' + (T monus T''), N, false) from R to O) .


*** "IGNORE" CCS DURING CC HOLDOFF PERIOD

*** During the holdoff timeout, the receiver stores the new GRTT if it
*** receives a new CC message.

rl [receivesCCduringHoldoff] :
  (outOfLink CC(T, T', N) from O to R)
  < R : RTTreceiver |
    ACKtimer : INF, ACKholdoffTimer : NZT >
  =>
  < R : RTTreceiver | GRTT : T' > .


*** IGNORE ACK AND NACK MESSAGES FROM OTHER RECEIVERS (BOTH NON-CLR AND CLR)

*** In the next two rules, the receiver ignores ACK and NACK messages,
*** either because it is the current limiting receiver, or because its
*** holdoff timer is running or it is not in a feedback cycle.

crl [ignoreACK] :
  (outOfLink ACK(T, N, B) from O to R)
  < R : RTTreceiver |
    ACKtimer : INF, ACKholdoffTimer : TI, CLR : B' >
  =>
  < R : RTTreceiver | >
  if (TI =/= 0) or (TI == INF) or (B' == true) .


crl [ignoreNACK1] :
  (outOfLink NACK(DUIL, T, B) from O to R)
  < R : RTTreceiverAlone |
    ACKtimer : INF, ACKholdoffTimer : TI, CLR : B' >
  =>
  < R : RTTreceiverAlone | >
  if (TI =/= 0) or (TI == INF) or (B' == true) .


*** A receiver ignores NACK messages if its feedback timer is running.

rl [ignoreNACK2] :
  (outOfLink NACK(DUIL, T, B) from O to R)
  < R : RTTreceiverAlone |
    ACKtimer : NZT, ACKholdoffTimer : INF >
  =>
  < R : RTTreceiverAlone | > .


*** When the holdoff timer expires, it is turned off.

rl [turnOffACKholdoffTimer] :
  < R : RTTreceiver | ACKholdoffTimer : 0 >
  =>
  < R : RTTreceiver | ACKholdoffTimer : INF > .


*** THE EFFECT OF TIME (THE DELTA FUNCTION)

eq delta(< S : RTTsenderAlone | clock : T, CCtransTimer : TI >, T') =
         < S : RTTsenderAlone | clock : T + T', CCtransTimer : TI monus T' > .
eq delta(< R : RTTreceiverAlone | clock : T, ACKtimer : TI,
                 ACKholdoffTimer : TI', nacksToBeSent : ML >, T') =
  < R : RTTreceiverAlone | clock : T + T', ACKtimer : TI monus T',
          ACKholdoffTimer : TI' monus T', nacksToBeSent : delta(ML, T') > .
  *** MAXIMAL POSSIBLE TIME ELAPSE (THE MTE FUNCTION)

  eq mte(< S : RTTsenderAlone | CCtransTimer : TI >) = TI .
  eq mte(< R : RTTreceiverAlone | ACKtimer : TI, ACKholdoffTimer : TI',
               nacksToBeSent : ML >) =
    if TI == INF and TI' == INF then INF
    else min(min(TI, TI'), leastDly(ML)) fi .

endtom)
```

B.3 The Data and Repair Transmission Specification

```
(tomod APPLICATIONS is
  inc MESSAGES .

  class SenderApplication | dataBuffer : MsgList .
  class ReceiverApplication | receiver : Oid, dataBuffer : DataUnitIdList .


  *** DELTA AND MTE FOR APPLICATIONS

  var A : Oid . var T : Time .

  eq delta(< A : SenderApplication | >, T) = < A : SenderApplication | > .
  eq mte(< A : SenderApplication | >) = INF .

  eq delta(< A : ReceiverApplication | >, T) = < A : ReceiverApplication | > .
  eq mte(< A : ReceiverApplication | >) = INF .

endtom)


(tomod NORM-DATA-TRANSMISSION is
  inc NAT-TIME-DOMAIN-WITH-INF .      inc TIMED-OO-SYSTEM .
  inc DETERMINISTIC-TICK-RULE .       inc TIMEOUT-FUNCTIONS .
  inc MESSAGES .                      inc NODES .
  inc NETWORK .                       inc APPLICATIONS .
  inc EXT-BOOL .

  vars A A' S R O : Oid .             var OS : OidSet .
  vars M M' : Msg .                   var DP DP' DP'' : DataPacket .
  vars ML ML' ML'' : MsgList .        vars B B' : Bool .
  vars DUI DUI' DUI'' : DataUnitId .  vars DUIL DUIL' DUIL'' : DataUnitIdList .
  vars TI TI' TI'' TI''' : TimeInf .  vars T T' : Time .
  vars N N' N'' N''' : Nat .
  vars NZN NZN' NZN'' NZN''' NZN'''' NZN''''' NZN'''''' : NzNat .




  *** THE SENDER ***

  class DTsender |
    dataBuffer : Msg,
    flushBuffer : Msg,
    repairTransmission : DataUnitIdList,
    currentTransPos : DataUnitId,
    lastNewDataId : DataUnitId,
    lastRepairDataId : DataUnitId,
    dataTransTimer : TimeInf,
    accNACKcontent : DataUnitIdList,
    invalidRepairRequests : DataUnitIdList,
    NACKaccumTimer : TimeInf,
    repairCycleHoldoffTimer : TimeInf,
    FLUSHtimer : TimeInf,
    FLUSHcounter : Nat,
    SQUELCHholdoffTimer : TimeInf .

  subclass DTsender < Sender .

  class DTsenderAlone . subclass DTsenderAlone < DTsender .


  *** DATA OBJECTS

  *** Usage: OBJECT(objectId, noOfSegsInObject, noOfSegsTransmitted)
  msg OBJECT : NzNat NzNat Nat -> Msg .

  *** An object block contains one or more data objects.
  *** Usage: objectBlock(listOfOBJECTs)
  msg objectBlock : MsgList -> Msg .
  msg noObjectBlock : -> Msg .


  *** TRANSMISSION OF NEW DATA

  *** The sender application gives the sender a new object block if its
  *** data buffer is empty.

  rl [nextObjectBlock] :
    < A : SenderApplication | dataBuffer : M ++ ML >
    < S : DTsender | dataBuffer : noObjectBlock >
    =>
    < A : SenderApplication | dataBuffer : ML >
    < S : DTsender | dataBuffer : M > .


  *** The sender transmits the first segment of the next object.

  crl [transmitFirstSegOfObject] :
    < S : DTsenderAlone |
      children : OS, GRTT : T,
      dataBuffer : objectBlock(ML ++ OBJECT(NZN, NZN’, 0) ++ ML’),
      lastNewDataId : N :: N’, sendRate : T’, dataTransTimer : TI,
      repairTransmission : nil >
    =>
    < S : DTsenderAlone |
      currentTransPos : s(N) :: 1,
      dataBuffer : objectBlock(ML ++ OBJECT(NZN, NZN’, 1) ++ ML’),
      lastNewDataId : s(N) :: 1, dataTransTimer : T’ >
    (multisend DATA(s(N) :: 1, NZN’, T, false) from S to OS)
    if s(N) == NZN /\ allSegsTransmitted(ML) /\ (TI == INF or TI == 0) .


  *** The sender transmits the rest of the current object.

  crl [transmitRestOfObject] :
    < S : DTsender |
      children : OS, GRTT : T,
      dataBuffer : objectBlock(ML' ++ OBJECT(NZN, NZN', NZN'') ++ ML''),
      lastNewDataId : NZN :: NZN'', sendRate : T', dataTransTimer : 0,
      repairTransmission : nil >
    =>
    < S : DTsender |
      currentTransPos : NZN :: s(NZN''),
      dataBuffer :
        objectBlock(ML' ++ OBJECT(NZN, NZN', s(NZN'')) ++ ML''),
      lastNewDataId : NZN :: s(NZN''),
      dataTransTimer :
        (if ML'' == nil and s(NZN'') == NZN' then INF else T' fi) >
    (multisend DATA(NZN :: s(NZN''), NZN', T, false) from S to OS)
    if NZN'' < NZN' .


  *** FLUSH PROCESS

  *** When the sender has finished transmitting its enqueued data content, it
  *** moves the data content to the flush buffer and sends the first FLUSH message.

  crl [initiateFlushProcess] :
   {< A : SenderApplication | dataBuffer : ML >
    < S : DTsender |
      children : OS, GRTT : T,
      dataBuffer : objectBlock(ML'), flushBuffer : noObjectBlock,
      FLUSHcounter : 0, FLUSHtimer : INF,
      NACKaccumTimer : INF, repairTransmission : nil >
    OC:ObjectConfiguration}
    =>
   {< A : SenderApplication | >
    < S : DTsender |
      dataBuffer : noObjectBlock, flushBuffer : objectBlock(ML'),
      FLUSHcounter : 1, FLUSHtimer : 2 * T >
    (multisend FLUSH(lastDataUnitId(objectBlock(ML')), T,
                     (if ML == nil then true else false fi))
     from S to OS)
    OC:ObjectConfiguration}
    if allSegsTransmitted(ML') .


  *** The sender transmits the rest of the FLUSH messages. After the final
  *** message is sent, it empties the flush buffer.

  crl [flushProcess] :
   {< A : SenderApplication | dataBuffer : ML >
    < S : DTsender |
      children : OS, NormRobustFactor : NZN, GRTT : T,
      dataBuffer : M, flushBuffer : M',
      FLUSHcounter : N, FLUSHtimer : 0,
      NACKaccumTimer : INF, repairTransmission : nil >
    OC:ObjectConfiguration}
    =>
   {< A : SenderApplication | >
    < S : DTsender |
      flushBuffer : (if s(N) == NZN then noObjectBlock else M' fi),
      FLUSHcounter : (if s(N) == NZN then 0 else s(N) fi),
      FLUSHtimer : (if s(N) == NZN then INF else 2 * T fi) >
    (multisend FLUSH(lastDataUnitId(M'), T,
      (if ML == nil and M == noObjectBlock then true else false fi))
     from S to OS)
    OC:ObjectConfiguration}
    if M' =/= noObjectBlock /\ N < NZN .


  *** NACK ACCUMULATION PERIOD

  *** When the sender receives a repair request, it starts its NACK
  *** accumulation period provided its is not already in a NACK accumulation
  *** period or its holdoff timer is running. It cancels the flush process
  *** if it is about to send a FLUSH message.

  crl [beginNACKaccumulation] :
    (outOfLink NACK(DUIL, T, B) from O to S)
    < S : DTsenderAlone |
      NormRobustFactor : NZN, GRTT : T',
      dataBuffer : M, flushBuffer : M',
      accNACKcontent : nil, NACKaccumTimer : INF,
      invalidRepairRequests : DUIL',
      repairCycleHoldoffTimer : TI >
    =>
    (if keepValidReqs(DUIL, M', M) == (nil).DataUnitIdList
     then
      *** there are no valid requests, so the sender doesn't
      *** start a NACK accumulation period
    < S : DTsenderAlone |
      invalidRepairRequests : addDUIlist(DUIL, DUIL') >
     else
     *** there are valid repair requests and the sender sets
     *** its NACKaccumTimer
    < S : DTsenderAlone |
      accNACKcontent : keepValidReqs(DUIL, M', M),
      NACKaccumTimer : (NZN + 1) * T',
      invalidRepairRequests :
        addDUIlist(keepInvalidReqs(DUIL, M', M), DUIL'),
      repairCycleHoldoffTimer : INF,
      FLUSHcounter : 0, FLUSHtimer : INF > fi)
    if TI == INF or TI == 0
    /\ not(M == noObjectBlock and M' == noObjectBlock) .


  *** The sender aggregates repair requests while the accumulation timer is
  *** running.

  crl [NACKaccumulationPeriod] :
    (outOfLink NACK(DUIL, T, B) from O to S)
    < S : DTsenderAlone |
      dataBuffer : M, flushBuffer : M',
      accNACKcontent : DUIL', NACKaccumTimer : TI,
      invalidRepairRequests : DUIL'' >
    =>
    < S : DTsenderAlone |
      accNACKcontent : addDUIlist(keepValidReqs(DUIL, M', M), DUIL'),
      invalidRepairRequests :
        addDUIlist(keepInvalidReqs(DUIL, M', M), DUIL'') >
    if TI =/= INF /\ TI =/= 0 .


  *** REPAIR TRANSMISSON

  *** When the repair request accumulation timer expires, the sender stops
  *** transmitting new data and prepares for repair transmission.

  rl [endOfNACKaccumulation] :
    < S : DTsender |
      GRTT : T, dataBuffer : M, flushBuffer : M',
      repairTransmission : DUIL, accNACKcontent : DUIL',
      NACKaccumTimer : 0, repairCycleHoldoffTimer : INF,
      sendRate : T', dataTransTimer : TI >
    =>
    < S : DTsender |
      repairTransmission :
        addDUIlist(replaceObjReq(DUIL', M, M'), DUIL),
      accNACKcontent : nil,
      NACKaccumTimer : INF, repairCycleHoldoffTimer : T,
      *** If TI = INF, then TI is set to the send rate,
      *** otherwise we keep the current value of TI:
      dataTransTimer : min(T', TI) > .


  *** Actual repair transmission starts in this rule. The sender transmits
  *** a DATA message for each requested data segment.

  rl [repairTransmission] :
    < A : SenderApplication | dataBuffer : ML >
    < S : DTsender |
      children : OS, GRTT : T, sendRate : T',
      dataBuffer : M, flushBuffer : M',
      repairTransmission : DUI ; DUIL,
      NormRobustFactor : NZN, dataTransTimer : 0, FLUSHtimer : INF >
      =>
    < A : SenderApplication | >
    < S : DTsender |
      currentTransPos : DUI, lastRepairDataId : DUI,
      repairTransmission : DUIL,
      dataTransTimer :
        (if ML == nil and allSegsTransmitted(M) and DUIL == nil
         then INF else T' fi),
      FLUSHtimer :
        (if M' =/= noObjectBlock and DUIL == nil then 0 else INF fi) >
    (multisend
     DATA(DUI, noOfSegs(DUI, M', M), T, true) from S to OS) .
```

```
*** HOLDOFF TIMEOUT AFTER NACK ACCUMULATION PERIOD

*** During the holdoff timeout, the sender keeps valid repair requests
*** from the receivers, but does not start a new repair cycle.

crl [NACKarrivesDuringHoldoffTimeout] :
  (outOfLink NACK(DUIL, T, B) from O to S)
  < S : DTsenderAlone |
    currentTransPos : DUI,
    dataBuffer : M, flushBuffer : M', repairTransmission : DUIL',
    repairCycleHoldoffTimer : TI, invalidRepairRequests : DUIL'' >
  =>
  < S : DTsenderAlone |
    repairTransmission :
      addDUIlist(keepForImmediateRepair(DUIL, DUI, M', M), DUIL'),
    invalidRepairRequests :
      addDUIlist(keepInvalidReqs(DUIL, M', M), DUIL'') >
  if TI =/= INF and TI =/= 0 .


*** When the holdoff timer expires, it is turned off.

rl [repairCycleHoldoffTimerExpires] :
  < S : DTsender | repairCycleHoldoffTimer : 0 >
  =>
  < S : DTsender | repairCycleHoldoffTimer : INF > .


*** SQUELCH MESSAGE

*** If the sender receives an invalid repair request, it sends a SQUELCH
*** to inform the receivers of which objects are valid for repair.

crl [sendSQUELCH] :
  < S : DTsender |
    children : OS, GRTT : T, dataBuffer : M, flushBuffer : M',
    invalidRepairRequests : DUIL, SQUELCHholdoffTimer : TI >
  =>
  < S : DTsender |
    invalidRepairRequests : nil, SQUELCHholdoffTimer : 2 * T >
  (if M' == noObjectBlock
   then (multisend SQUELCH(firstDataUnitId(M), T) from S to OS)
   else (multisend SQUELCH(firstDataUnitId(M'), T) from S to OS)
   fi)
  if DUIL =/= nil /\ (TI == 0 or TI == INF) .


*** When the SQUELCH holdoff timer expires, it is turned off, and
*** the sender is free to send a new SQUELCH for invalid repair requests.

rl [SQUELCHholdoffTimerExpires] :
  < S : DTsender | SQUELCHholdoffTimer : 0 >
  =>
  < S : DTsender | SQUELCHholdoffTimer : INF > .
```

```
*** THE EFFECT OF TIME

eq delta(< S : DTsenderAlone | clock : T, dataTransTimer : TI,
                 NACKaccumTimer : TI’, repairCycleHoldoffTimer : TI’’,
                 FLUSHtimer : TI’’’ >, T’) =
         < S : DTsenderAlone | clock : T + T’, dataTransTimer : TI monus T’,
                 NACKaccumTimer : TI’ monus T’,
                 repairCycleHoldoffTimer : TI’’ monus T’,
                 FLUSHtimer : TI’’’ monus T’ > .


*** MAXIMUM TIME ELAPSE

eq mte(< S : DTsenderAlone | dataTransTimer : TI, NACKaccumTimer : TI’,
               repairCycleHoldoffTimer : TI’’, FLUSHtimer : TI’’’ >) =
       min(TI, min(TI’, min(TI’’, TI’’’))) .



*** THE RECEIVER ***

class DTreceiver |
  receiveBuffer : MsgList,
  repairNeeds : DataUnitIdList,
  sendersCurrTransPos : DataUnitId,
  nextExpectedDUI : DataUnitId,
  repairRequests : DataUnitIdList,
  extRepRequests : DataUnitIdList,
  NACKbackoffTimer : TimeInf,
  NACKcycleHoldoffTimer : TimeInf .

subclass DTreceiver < Receiver .

class DTreceiverAlone . subclass DTreceiverAlone < DTreceiver .


*** RECEPTION AND FORWARDING OF DATA, INITIATION OF NACK
*** CYCLE AT OBJECT BOUNDARY

*** The receiver receives the next expected data segment.

rl [receivesTheNextExpectedSegment] :
  (outOfLink DATA(NZN :: NZN’, NZN’’, T, false) from O to R)
  < R : DTreceiver |
    receiveBuffer : ML, nextExpectedDUI : NZN :: NZN’ >
  =>
  < R : DTreceiver |
    GRTT : T,
    receiveBuffer : ML ++ DATA(NZN :: NZN’, NZN’’, T, false),
    sendersCurrTransPos : NZN :: NZN’,
    nextExpectedDUI : (if NZN’ < NZN’’ then (NZN :: s(NZN’))
      else (s(NZN) :: 1) fi) > .


*** The received segment is not the expected one, and the receiver sets
*** its backoff timer, if possible.
```

```
crl [gapInReception] :
  (outOfLink DATA(DUI, NZN, T, false) from O to R)
  < R : DTreceiver |
    randomSeed : N, NormRobustFactor : NZN', gsize : NZN'',
    receiveBuffer : ML, nextExpectedDUI : DUI',
    repairNeeds : DUIL, repairRequests : DUIL',
    NACKbackoffTimer : TI, NACKcycleHoldoffTimer : TI' >
  =>
  (if objectBoundary(DUI, ML)
   and TI == INF and (TI' == 0 or TI' == INF)
   then *** NACK cycle is triggered at an object boundary
  < R : DTreceiver |
    randomSeed : random(N), GRTT : T,
    receiveBuffer : ML ++ DATA(DUI, NZN, T, false),
    sendersCurrTransPos : DUI,
    repairNeeds : DUIL ; recordRepairNeeds(DUI, DUI', ML),
    repairRequests :
      addDUIlist(DUIL,
                 (DUIL' ; recordRepairNeeds(DUI, DUI', ML))),
    NACKbackoffTimer : randomBackoff(random(N), NZN', T, NZN'') >
   else *** the seg. and info. on missing segments is simply stored
  < R : DTreceiver |
    GRTT : T,
    receiveBuffer : (if largerThan(DUI, DUI')
                      then ML ++ DATA(DUI, NZN, T, false)
                      else ML fi), *** duplicates are ignored
    sendersCurrTransPos : DUI,
    repairNeeds : DUIL ; recordRepairNeeds(DUI, DUI', ML) > fi)
  if DUI =/= DUI' .


*** The receiver receives a repair segment. If it is about to request a
*** repair for this segment, it removes the repair request.

rl [receivesRepairSegment] :
  (outOfLink DATA(DUI, NZN, T, true) from O to R)
  < R : DTreceiver |
    receiveBuffer : ML, nextExpectedDUI : DUI',
    repairNeeds : DUIL, repairRequests : DUIL' >
  =>
  if missingSeg(DUI, DUIL) *** this is a missing segment
  then
  < R : DTreceiver |
    GRTT : T, sendersCurrTransPos : DUI,
    receiveBuffer : addRepairMsg(DATA(DUI, NZN, T, true), ML),
    repairNeeds : removeRepReq(DATA(DUI, NZN, T, true), DUIL),
    repairRequests : removeRepReq(DATA(DUI, NZN, T, true), DUIL'),
    nextExpectedDUI : (if DUI == DUI' then
      newNextExpectedDUI(
        recBuffFromDUI(addRepairMsg(DATA(DUI, NZN, T, true), ML), DUI))
      else DUI' fi) >
  else *** the message is ignored
  < R : DTreceiver | GRTT : T > fi .


*** The receiver forwards data segments to its application. It keeps the
*** most recently received DATA message in case it needs to ask for repair
*** as the message contains necessary information.

crl [forwardDataSegments] :
  < A : ReceiverApplication | receiver : R, dataBuffer : DUIL >
  < R : DTreceiver |
    receiveBuffer : DATA(NZN :: NZN', NZN'', T, B) ++ ML,
    nextExpectedDUI : N :: N' >
  =>
  < A : ReceiverApplication | dataBuffer : DUIL ; (NZN :: NZN') >
  < R : DTreceiver | receiveBuffer : ML >
  if (not(NZN' == NZN'' and N' == 1 and s(NZN) == N)
    and not(NZN :: s(NZN') == N :: N'))
  or N :: N' == (0 :: 0) .


*** The receiver forwards the very last data segment.

rl [forwardLastDataSegment] :
  < A : ReceiverApplication | receiver : R, dataBuffer : DUIL >
  < R : DTreceiver | receiveBuffer : DATA(NZN :: NZN', NZN', T, B) >
  < S : DTsender | dataBuffer : noObjectBlock, flushBuffer : noObjectBlock >
  < A' : SenderApplication | dataBuffer : (nil).MsgList >
  =>
  < A : ReceiverApplication | dataBuffer : DUIL ; (NZN :: NZN') >
  < R : DTreceiver | receiveBuffer : (nil).MsgList, nextExpectedDUI : 0 :: 0 >
  < S : DTsender | > < A' : SenderApplication | > .


*** INITIATION OF NACK CYCLE UPON RECEPTION OF FLUSH

*** The receiver receives a FLUSH message and initiates a repair request
*** if it needs repair.

crl [NACKcycleInitiatedByFLUSH] :
  (outOfLink FLUSH(NZN :: NZN', T, B) from O to R)
  < R : DTreceiver |
    randomSeed : N, NormRobustFactor : NZN'', gsize : NZN''',
    receiveBuffer : ML, nextExpectedDUI : DUI,
    repairNeeds : DUIL, repairRequests : DUIL',
    NACKbackoffTimer : TI, NACKcycleHoldoffTimer : TI',
    sendersCurrTransPos : DUI' >
  =>
  (if (keepValidNACKcontent(DUIL, NZN :: NZN') =/= nil
   or recordRepairNeeds(s(NZN) :: 1, DUI,
                        recBuffUpToDUI(ML, s(NZN) :: 1)) =/= nil)
   and TI == INF and (TI' == INF or TI' == 0)
   then
   < R : DTreceiver |
     GRTT : T, randomSeed : random(N),
     repairNeeds : addDUIlist(recordRepairNeeds(s(NZN) :: 1, DUI,
                                  recBuffUpToDUI(ML, s(NZN) :: 1)),
                              DUIL),
     repairRequests : addDUIlist(recordRepairNeeds(s(NZN) :: 1, DUI,
                                     recBuffUpToDUI(ML, s(NZN) :: 1)),
                                 keepValidNACKcontent(DUIL, NZN :: NZN')),
     NACKbackoffTimer : randomBackoff(random(N), NZN'', T, NZN'''),
     sendersCurrTransPos : (if B then 0 :: 0 else DUI' fi) >
   else
   < R : DTreceiver |
     GRTT : T,
     nextExpectedDUI :
       (if B == true and DUI == s(NZN) :: 1 then 0 :: 0 else DUI fi) > fi)
  if largerThan(s(NZN) :: 1, DUI) /\ DUI =/= 0 :: 0 .


*** A receiver which has no repair needs up to the sequence number
*** identified by the FLUSH message, ignores the FLUSH.

crl [ignoreFLUSH] :
  (outOfLink FLUSH(DUI, T, B) from O to R)
  < R : DTreceiver | nextExpectedDUI : DUI' >
  =>
  < R : DTreceiver | GRTT : T >
  if smallerThan(DUI, DUI') or DUI' == 0 :: 0 .


*** EXTERNAL REPAIR STATE ACCUMULATION

*** The receiver accumulates external repair requests while its backoff
*** timer is running. Otherwise, NACK messages are ignored.

rl [accumulateExternalRepairState] :
  (outOfLink NACK(DUIL, T, B) from O to R)
  < R : DTreceiver |
    extRepRequests : DUIL', NACKbackoffTimer : TI >
  =>
  if TI =/= INF and TI =/= 0
  then
  < R : DTreceiver |
    extRepRequests : addDUIlist(DUIL, DUIL') >
  else
  < R : DTreceiver | > fi .


*** TRANSMISSION OF NACK FOLLOWED BY HOLDOFF TIMEOUT

*** When its backoff timer expires, the receiver sends a repair request
*** provided it still has repair needs.

crl [makeNACKdecision] :
  < R : DTreceiverAlone |
    parent : O, NormRobustFactor : NZN, GRTT : T,
    CLR : B, sendersCurrTransPos : DUI,
    repairRequests : DUIL, extRepRequests : DUIL',
    NACKbackoffTimer : 0, NACKcycleHoldoffTimer : INF >
  =>
  (if not(alreadyNACKed(DUIL, DUIL')) and
     (DUI == 0 :: 0 or-else smallerThan(first(DUIL), DUI))
   then
   (< R : DTreceiverAlone |
      repairRequests : nil, extRepRequests : nil,
      NACKbackoffTimer : INF, NACKcycleHoldoffTimer : (NZN + 2) * T >
   (intoLink NACK(sublistUpToDUI(DUIL, DUI), 0, B) from R to O))
   else
   < R : DTreceiverAlone |
    repairRequests : nil, extRepRequests : nil,
    NACKbackoffTimer : INF > fi)
  if DUIL =/= nil .


*** The receiver turns of the backoff timer if it has no repair needs.

rl [noRepairRequests] :
  < R : DTreceiver | NACKbackoffTimer : 0, repairRequests : nil >
  =>
  < R : DTreceiver | NACKbackoffTimer : INF > .


*** The repair request holdoff timer expires, and is turned off.

rl [turnOffNACKcycleHoldoffTimer] :
  < R : DTreceiver | NACKcycleHoldoffTimer : 0 >
  =>
  < R : DTreceiver | NACKcycleHoldoffTimer : INF > .


*** CANCEL PENDING INVALID REPAIR REQUESTS

*** If it receives a SQUELCH, the receiver removed invalid repair requests.

rl [removeInvalidRepairRequests] :
  (outOfLink SQUELCH(DUI, T) from O to R)
  < R : DTreceiver |
    receiveBuffer : ML, nextExpectedDUI : DUI',
    repairNeeds : DUIL, repairRequests : DUIL' >
  =>
  < R : DTreceiver |
    GRTT : T, repairNeeds : removeSquelchedRepReqs(DUIL, DUI),
    repairRequests : removeSquelchedRepReqs(DUIL', DUI),
    nextExpectedDUI : (if smallerThan(DUI', DUI)
      then newNextExpectedDUI(recBuffFromDUI(ML, DUI))
      else DUI' fi) > .


*** THE EFFECT OF TIME

eq delta(< R : DTreceiverAlone | clock : T, NACKbackoffTimer : TI,
                 NACKcycleHoldoffTimer : TI' >, T') =
         < R : DTreceiverAlone | clock : T + T',
                 NACKbackoffTimer : TI monus T',
                 NACKcycleHoldoffTimer : TI' monus T' > .


*** MAXIMUM TIME ELAPSE

eq mte(< R : DTreceiverAlone | NACKbackoffTimer : TI,
               NACKcycleHoldoffTimer : TI' >) =
       min(TI, TI') .


*** FUNCTIONS USED BY THE SENDER AND RECEIVER
*** Are all segments of an object or a list of objects
*** in the sender's dataBuffer transmitted?

op allSegsTransmitted : Msg -> Bool .
op allSegsTransmitted : MsgList -> Bool .
eq allSegsTransmitted(noObjectBlock) = true .
eq allSegsTransmitted(objectBlock(ML)) = allSegsTransmitted(ML) .
eq allSegsTransmitted((nil).MsgList) = true .
eq allSegsTransmitted(OBJECT(NZN, NZN', N) ++ ML) =
  NZN' == N and allSegsTransmitted(ML) .


*** Find the smallest data unit id in an object block.

op firstDataUnitId : Msg -> DataUnitId .
eq firstDataUnitId(objectBlock(OBJECT(NZN, NZN', N) ++ ML)) = NZN :: 1 .


*** Find the greatest data unit id in an object block

op lastDataUnitId : Msg -> DataUnitId .
eq lastDataUnitId(objectBlock(ML ++ OBJECT(NZN, NZN', N))) = NZN :: NZN' .


*** Is the first data unit id smaller than the second?

op smallerThan : DataUnitId DataUnitId -> Bool .
eq smallerThan(NZN :: N, NZN' :: N') =
  NZN < NZN' or (NZN == NZN' and N < N') .


*** Is the first data unit id larger than the second?

op largerThan : DataUnitId DataUnitId -> Bool .
eq largerThan(NZN :: N, NZN' :: N') =
  NZN > NZN' or (NZN == NZN' and N > N') .


*** Add one single data unit id to a sorted list of ids.

op addDUI : DataUnitId DataUnitIdList -> DataUnitIdList .
eq addDUI(NZN :: N, nil) = NZN :: N .
eq addDUI(DUI, (DUI' ; DUIL)) =
  *** already in the list:
  if DUI == DUI'
  then (DUI' ; DUIL)
  *** add to the sorted list:
  else (if smallerThan(DUI, DUI')
        then DUI ; (DUI' ; DUIL)
        else DUI' ; addDUI(DUI, DUIL) fi)
  fi .


*** Add a list of new data unit ids to a sorted list and remove doubles.
```

```
op addDUIlist : DataUnitIdList DataUnitIdList -> DataUnitIdList .
eq addDUIlist(nil, DUIL) = DUIL .
eq addDUIlist(DUI ; DUIL, DUIL') = addDUIlist(DUIL, addDUI(DUI, DUIL')) .


*** Find the number of segments in the object denoted
*** by the dataUnitId.

op noOfSegs : DataUnitId Msg Msg -> Nat .
op findNoOfSegs : DataUnitId Msg -> Nat .
eq noOfSegs(DUI, M, M') = findNoOfSegs(DUI, M) + findNoOfSegs(DUI, M') .
eq findNoOfSegs(DUI, noObjectBlock) = 0 .
eq findNoOfSegs(DUI, objectBlock(nil)) = 0 .
eq findNoOfSegs(NZN :: N, objectBlock(OBJECT(NZN', NZN'', N') ++ ML)) =
  if NZN == NZN' then NZN''
  else findNoOfSegs(NZN :: N, objectBlock(ML)) fi .


*** Is a repair request valid, i.e. does it concerns a data
*** segment that is either in the flush buffer or data buffer?
*** Usage: valid(repReq, objectBlock, objectBlock)

op valid : DataUnitId Msg Msg -> Bool .
op validInObjBlock : DataUnitId Msg -> Bool .
eq valid(DUI, M, M') = validInObjBlock(DUI, M) or validInObjBlock(DUI, M') .
eq validInObjBlock(DUI, noObjectBlock) = false .
eq validInObjBlock(DUI, objectBlock(nil)) = false .
eq validInObjBlock(NZN :: N, objectBlock(OBJECT(NZN', NZN'', N') ++ ML)) =
  if NZN == NZN' and N <= NZN''
  then true
  else validInObjBlock(NZN :: NZN', objectBlock(ML)) fi .


*** Keep repair requests which are valid for immediate repair.
*** Usage: keepForImmediateRepair
***   (repairRequests, currTransPos, objectBlock, objectBlock)

op keepForImmediateRepair : DataUnitIdList DataUnitId Msg Msg ->
  DataUnitIdList .
eq keepForImmediateRepair(nil, DUI, M, M') = nil .
eq keepForImmediateRepair((NZN :: N) ; DUIL, NZN' :: NZN'', M, M') =
  if valid(NZN :: N, M, M') and largerThan(NZN :: N, NZN' :: NZN'')
  then ((NZN :: N) ; keepForImmediateRepair(DUIL, NZN' :: NZN'', M, M'))
  else (if valid(NZN :: N, M, M') and NZN == NZN' and N  == 0
        then generateDUIsequence(NZN :: s(NZN''),
                                 NZN :: s(noOfSegs(NZN :: N, M, M'))) ;
             keepForImmediateRepair(DUIL, NZN' :: NZN'', M, M')
        else keepForImmediateRepair(DUIL, NZN' :: NZN'', M, M') fi) fi .


*** Keep repair requests which are valid for repair, i.e.
*** requests for data which the sender has in its data
*** or flush buffer.
*** Usage: keepValidReqs(repairRequests, objectBlock, objectBlock)

op keepValidReqs : DataUnitIdList Msg Msg -> DataUnitIdList .
eq keepValidReqs(nil, M, M') = nil .
eq keepValidReqs(DUI ; DUIL, M, M’) =
  if valid(DUI, M, M’)
  then (DUI ; keepValidReqs(DUIL, M, M’))
  else keepValidReqs(DUIL, M, M’) fi .


*** Keep repair requests which are out of range/invalid.
*** Usage: keepInvalidReqs(repairRequests, objectBlock, objectBlock)

op keepInvalidReqs : DataUnitIdList Msg Msg -> DataUnitIdList .
eq keepInvalidReqs(nil, M, M’) = nil .
eq keepInvalidReqs(DUI ; DUIL, M, M’) =
  if valid(DUI, M, M’)
  then keepInvalidReqs(DUIL, M, M’)
  else (DUI ; keepInvalidReqs(DUIL, M, M’)) fi .


*** keep all DUIs that are valid for repair according to
*** the lastValidRepairPosition
*** Usage: keepValidNACKcontent(listOfRepairNeeds, lastValidRepairPosition)

op keepValidNACKcontent : DataUnitIdList DataUnitId -> DataUnitIdList .
eq keepValidNACKcontent(nil, DUI) = nil .
eq keepValidNACKcontent(DUI ; DUIL, DUI’) =
  if smallerThan(DUI, DUI’) or DUI == DUI’
  then (DUI ; keepValidNACKcontent(DUIL, DUI’))
  else keepValidNACKcontent(DUIL, DUI’) fi .


*** Do external repair requests equal or supersede the
*** receiver’s repair requests? I.e., have all the elements
*** in the receiver’s list been NACKed by others?
*** Usage: alreadyNACKed(recRepRequests, extRepRequests)

op alreadyNACKed : DataUnitIdList DataUnitIdList -> Bool .
eq alreadyNACKed(nil, DUIL) = true .
eq alreadyNACKed(DUI ; DUIL, DUIL’) =
  if inDUIlist(DUI, DUIL’)
  then alreadyNACKed(DUIL, DUIL’)
  else false fi .


*** Return the sublist of a data unit id list from lowest
*** element up to a certain DUI.
*** Usage: sublistUpToDUI(DUIlist, upToThisDUI)

op sublistUpToDUI : DataUnitIdList DataUnitId -> DataUnitIdList .
eq sublistUpToDUI(nil, DUI) = nil .
eq sublistUpToDUI(DUIL, 0 :: 0) = DUIL .
eq sublistUpToDUI(DUI ; DUIL, DUI’) =
  if smallerThan(DUI, DUI’)
  then DUI ; sublistUpToDUI(DUIL, DUI’)
  else nil fi .


*** Has the reception reached an object boundary, i.e.
*** has a segment of the next object arrived?
*** Usage: objectBoundary(receivedSeg, recBuffer)

op objectBoundary : DataUnitId MsgList -> Bool .
eq objectBoundary(DUI, nil) = false .
eq objectBoundary(NZN :: NZN', ML ++ DATA(NZN'' :: NZN''', NZN'''', T, B)) =
  NZN'' < NZN .


*** Generate data unit ids for the missing segments between
*** the last segment in the receiveBuffer (or NextExpectedDUI,
*** in case this is the start of transmission and the first
*** segment(s) is lost) and the segment which just arrived.
*** If an entire object is missing, a DUI (n :: 0) is
*** generated for that object.
*** Usage: recordRepairNeeds(receivedSeg, nextExpectedDUI, recBuff)

op recordRepairNeeds : DataUnitId DataUnitId MsgList ->
  DataUnitIdList .

*** The two equations handle the cases where, respectively,
*** there is information in the receiveBuffer about the number
*** of segments in an object, and there is no such information.

eq recordRepairNeeds(NZN :: NZN', NZN'' :: NZN''',
  ML ++ DATA(NZN'''' :: NZN''''', NZN'''''', T, B)) =
  if NZN'''' < NZN
  then generateDUIsequence(NZN'''' :: s(NZN'''''),
                           NZN'''' :: s(NZN'''''')) ;
       recordRepairNeeds(NZN :: NZN', s(NZN'''') :: 1, nil)
  else generateDUIsequence(NZN'''' :: s(NZN'''''),
                           NZN :: NZN') fi .
eq recordRepairNeeds(NZN :: NZN', NZN'' :: NZN''', nil) =
  if NZN'' < NZN
  then (NZN'' :: 0) ;
        recordRepairNeeds(NZN :: NZN', s(NZN'') :: 1, nil)
  else generateDUIsequence(NZN'' :: NZN''', NZN :: NZN') fi .


*** Generate DUIs from and including first argument
*** to (not including) last argument in the same object.
*** Usage: generateDUIsequence(fromDUI, toDUI)

op generateDUIsequence : DataUnitId DataUnitId -> DataUnitIdList .
eq generateDUIsequence(NZN :: NZN', NZN :: NZN'') =
  if NZN' == NZN''
  then (nil).DataUnitIdList
  else (NZN :: NZN') ;
        generateDUIsequence(NZN :: s(NZN'), NZN :: NZN'') fi .


*** Remove the request for the received segment. If the entire
*** object is missing, replace the object request with
*** segment requests for the remaining segments.
*** Usage: removeRepReq(receivedMsg, repairNeeds/Requests)

op removeRepReq : DataPacket DataUnitIdList -> DataUnitIdList .
eq removeRepReq(DP, (nil).DataUnitIdList) = (nil).DataUnitIdList .
eq removeRepReq(DATA(DUI, NZN, T, B), (NZN' :: NZN'') ; DUIL) =
  if DUI == NZN' :: NZN''
  then DUIL
  else (NZN' :: NZN'') ; removeRepReq(DATA(DUI, NZN, T, B), DUIL) fi .
eq removeRepReq(DATA(NZN :: NZN', NZN'', T, B), (NZN :: 0) ; DUIL) =
  if NZN' == 1
  then (generateDUIsequence(NZN :: s(NZN'), NZN :: s(NZN'')) ; DUIL)
  else (if NZN' == NZN''
        then (generateDUIsequence(NZN :: 1, NZN :: NZN') ; DUIL)
        else (generateDUIsequence(NZN :: 1, NZN :: NZN') ;
  generateDUIsequence(NZN :: s(NZN'), NZN :: s(NZN'')) ; DUIL) fi) fi .


*** Update nextExpectedDUI by running through the receiveBuffer
*** and finding the next gap, if any.
*** Usage: newNextExpectedDUI(receiveBuffer)

op newNextExpectedDUI : MsgList -> DataUnitId .
eq newNextExpectedDUI(DATA(NZN :: NZN', NZN'', T, B) ++ (nil).MsgList) =
  if NZN' < NZN''
  then (NZN :: s(NZN'))
  else (s(NZN) :: 1) fi .
eq newNextExpectedDUI(DATA(NZN :: NZN', NZN'', T, B) ++
  DATA(NZN''' :: NZN'''', NZN''''', T', B') ++ ML) =
  if gap(DATA(NZN :: NZN', NZN'', T, B),
         DATA(NZN''' :: NZN'''', NZN''''', T', B'))
  then (if NZN' < NZN'' then (NZN :: s(NZN'))
        else (s(NZN) :: 1) fi)
  else
  newNextExpectedDUI(DATA(NZN''' :: NZN'''', NZN''''', T', B') ++ ML) fi .


*** Is there a gap between two DATA messages?

op gap : DataPacket DataPacket -> Bool .
eq gap(DATA(NZN :: NZN', NZN'', T, B),
  DATA(NZN''' :: NZN'''', NZN''''', T', B')) =
  not((NZN == NZN''' and s(NZN') == NZN'''') or
  (NZN' == NZN'' and s(NZN) == NZN''' and NZN'''' == 1)) .


*** Is this segment among the missing ones?
*** Usage: missingSeg(repairSeg, repairNeeds)

op missingSeg : DataUnitId DataUnitIdList -> Bool .
eq missingSeg(DUI, DUIL) =
  inDUIlist(DUI, DUIL) or matchesObjectRequest(DUI, DUIL) .


*** Is the data unit id in the DUI list?

op inDUIlist : DataUnitId DataUnitIdList -> Bool .
eq inDUIlist(DUI, (nil).DataUnitIdList) = false .
eq inDUIlist(DUI, DUI' ; DUIL) = DUI == DUI' or inDUIlist(DUI, DUIL) .


*** Is there a request for the entire object to which
*** the segment belongs in the data unit id list?

op matchesObjectRequest : DataUnitId DataUnitIdList -> Bool .
eq matchesObjectRequest(DUI, (nil).DataUnitIdList) = false .
eq matchesObjectRequest(NZN :: NZN', (NZN'' :: N) ; DUIL) =
  (NZN == NZN'' and N == 0) or matchesObjectRequest(NZN :: NZN', DUIL) .


*** Add repair message to sorted receiveBuffer.
*** Usage: addRepairMsg(repairMsg, receiveBuffer)

op addRepairMsg : DataPacket MsgList -> MsgList .
eq addRepairMsg(DP, (nil).MsgList) = DP .
eq addRepairMsg(DP, DP' ++ ML) =
  if smallerThan(dataUnitId(DP), dataUnitId(DP'))
  then DP ++ DP' ++ ML
  else (if ML == nil
        then DP' ++ DP
        else DP' ++ addRepairMsg(DP, ML) fi) fi .


*** Returns the data unit id in a DATA message

op dataUnitId : DataPacket -> DataUnitId .
eq dataUnitId(DATA(NZN :: NZN', NZN'', T, B)) = NZN :: NZN' .


*** Removes the repair requests denoted as invalid by
*** a SQUELCH message, that is, all requests for segments or
*** objects with ids lower than the earliest valid repair position.
*** Usage: removeSquelchedRepReqs(DUIlist, startDUI)

op removeSquelchedRepReqs : DataUnitIdList DataUnitId -> DataUnitIdList .
eq removeSquelchedRepReqs((nil).DataUnitIdList, DUI) = (nil).DataUnitIdList .
eq removeSquelchedRepReqs((NZN :: N) ; DUIL, NZN' :: NZN'') =
  if smallerThan(NZN :: N, NZN' :: NZN'') and
  not(NZN == NZN' and N == 0)
  then removeSquelchedRepReqs(DUIL, NZN' :: NZN'')
  else (NZN :: N) ; DUIL fi .


*** Returns the messages in the receiveBuffer that have an id
*** equal to or larger than the given DUI.
*** Usage: recBuffFromDUI(recBuff, earliestValidRepPos)

op recBuffFromDUI : MsgList DataUnitId -> MsgList .
eq recBuffFromDUI((nil).MsgList, DUI) = (nil).MsgList .
eq recBuffFromDUI(DATA(DUI, NZN, T, B) ++ ML, DUI') =
  if DUI == DUI' or largerThan(DUI, DUI')
  then DATA(DUI, NZN, T, B) ++ ML
  else recBuffFromDUI(ML, DUI') fi .


*** Returns the messages in the receiveBuffer that have an id
*** smaller than the given DUI.
*** Usage: recBuffUpToDUI(recBuff, dataUnitId)
```

```
  op recBuffUpToDUI : MsgList DataUnitId -> MsgList .
  eq recBuffUpToDUI((nil).MsgList, DUI) = (nil).MsgList .
  eq recBuffUpToDUI(DATA(DUI, NZN, T, B) ++ ML, DUI') =
    if smallerThan(DUI, DUI')
    then DATA(DUI, NZN, T, B) ++ recBuffUpToDUI(ML, DUI')
    else nil fi .


  *** Return the first element in a data unit id list.

  op first : DataUnitIdList -> DataUnitId .
  eq first((nil).DataUnitIdList) = (nil).DataUnitIdList .
  eq first(DUI ; DUIL) = DUI .


  *** Run through a list of repair requests and replace
  *** any object requests with segment requests for the entire object.
  *** Usage: replaceObjReq(repReqList, objectBlock, objectBlock)

  op replaceObjReq : DataUnitIdList Msg Msg -> DataUnitIdList .
  eq replaceObjReq((nil).DataUnitIdList, M, M') = (nil).DataUnitIdList .
  eq replaceObjReq((NZN :: N) ; DUIL, M, M') =
    if N == 0
    then generateDUIsequence(NZN :: 1,
                             NZN :: s(noOfSegs(NZN :: N, M, M'))) ;
         replaceObjReq(DUIL, M, M')
    else (NZN :: N) ; replaceObjReq(DUIL, M, M') fi .

endtom)
```

B.4 The Combined Specification

```
(tomod NORM-COMBINED is
  including NORM-RTT .
  including NORM-DATA-TRANSMISSION .

  class SenderCombined .
  subclass SenderCombined < DTsender .
  subclass SenderCombined < RTTsender .

  class ReceiverCombined .
  subclass ReceiverCombined < DTreceiver .
  subclass ReceiverCombined < RTTreceiver .

  vars S R : Oid .
  vars T T' : Time .
  vars TI TI' TI'' TI''' TI'''' TI''''' : TimeInf .


  *** DELTA AND MTE FOR THE COMBINED PROTOCOL

  eq delta(< S : SenderCombined | clock : T, dataTransTimer : TI,
                 NACKaccumTimer : TI', repairCycleHoldoffTimer : TI'',
                 FLUSHtimer : TI''', SQUELCHholdoffTimer : TI'''',
                 CCtransTimer : TI''''' >, T') =
  < S : SenderCombined | clock : T + T', dataTransTimer : TI monus T',
          NACKaccumTimer : TI' monus T', repairCycleHoldoffTimer : TI'' monus T',
          FLUSHtimer : TI''' monus T', SQUELCHholdoffTimer : TI'''' monus T',
          CCtransTimer : TI''''' monus T' > .

  eq delta(< R : ReceiverCombined | clock : T, NACKbackoffTimer : TI,
                   NACKcycleHoldoffTimer : TI', ACKtimer : TI'',
                   ACKholdoffTimer : TI''' >, T') =
  < R : ReceiverCombined | clock : T + T', NACKbackoffTimer : TI monus T',
          NACKcycleHoldoffTimer : TI' monus T', ACKtimer : TI'' monus T',
          ACKholdoffTimer : TI''' monus T' > .


  eq mte(< S : SenderCombined | dataTransTimer : TI, NACKaccumTimer : TI',
                   repairCycleHoldoffTimer : TI'', FLUSHtimer : TI''',
                   SQUELCHholdoffTimer : TI'''', CCtransTimer : TI''''' >) =
  min(min(min(min(min(TI, TI'), TI''), TI'''), TI''''), TI''''') .

  eq mte(< R : ReceiverCombined | NACKbackoffTimer : TI,
                   NACKcycleHoldoffTimer : TI', ACKtimer : TI'',
                   ACKholdoffTimer : TI''' >) =
  min(min(min(TI, TI'), TI''), TI''') .

endtom)
```