A Classification of Total Order Specifications and its Application to Fixed Sequencer-based Implementations*

Roberto Baldoni, Stefano Cimmino, Carlo Marchetti
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, 00198, Roma, Italy
email: {baldoni,cimmino,marchet}@dis.uniroma1.it

Abstract

During the last two decades the design and development of total order (TO) communications has been one of the main research topics in dependable distributed computing. The huge amount of research work has produced several TO specifications and a wide variety of TO implementations with different guarantees whose differences are often left hidden or unclear. This paper presents a systematic classification of six distinct TO specifications based on a well-defined formal framework. The classification allows us (i) to define in a formal way the differences among the behaviors of faulty and correct processes admitted by each specification, and (ii) to easily match TO implementations with respect to their enforced specification. The classification is applied to study the properties of eight variations of TO implementations based on a fixed sequencer given in a well-known context, namely primary component group communication systems.

1 Introduction

Since Lamport's seminal paper [22], the problem of *totally ordered (TO) communications*, also called *atomic communications*, has been extensively studied in the literature. Intuitively, a TO primitive ensures that processes of a message-passing distributed system deliver the same sequence of messages. Researchers have pointed out several application scenarios in which the use of TO communications is extremely useful, for example maintaining consistency among the internal states of a set of deterministic replicas (active software replication [26]) and facing the problem of delivering stock quote information in the same order to a set of stock operators while preserving fairness of exchanges [11].

During the last two decades more than sixty TO implementations have been designed [16]. Typically, these ensure that, in the absence of failures, processes deliver the same set of messages in the same order. In contrast, distinct TO implementations may behave in different manners in the presence of failures. As an example, some TO implementations allow faulty processes to deliver messages in an order different from correct ones. This aspect (and especially its impact on applications) still deserves careful analysis. Let us consider an application logic based on

*This work is partially supported by a grant from EU IST Project "EU-PUBLI.COM" (#IST-2001-35217), and by a grant from MIUR in the context of project "MAIS".

active replication with strong replica consistency requirements [18]. To preserve safety (i.e., strong replica consistency), it is necessary that *all* replicas (faulty or not) deliver the same sequence of messages. Therefore the occurrence of wrong message ordering at a faulty replica would violate safety if not appropriately handled by the application logic.[1] Unfortunately, the analysis of the TO implementation's behavior under failure scenarios is often underestimated, especially by practitioners. Furthermore, even if one wishes to undertake such an analysis, he/she has to face the problem of understanding a complex (or even missing) specification, e.g. [9, 21]. This problem is also reinforced by the fact that specifications are sometimes given using different ad-hoc formalisms, which are difficult to compare among themselves, e.g. [17, 21]. Despite several valuable works have tried to position the existing literature into a uniform comprehensive framework (e.g. [14, 16]), the problem of clarifying in a simple and easy way both the meaning of TO specifications and its impact on applications has not been addressed in a satisfactory way.

This paper aims to fill this need. Using a formal framework, based on first order logic as specification language, we present and describe in a systematic way eight existing TO specifications. Some of these specifications appeared in the literature, while others have been derived from our formal analysis. Besides the formalization of TO specifications, the novelty of the analysis lies in the explicit description of failure scenarios admitted by each specification, in terms of the possible behavior of faulty processes. This permits a deep understanding of the impact of each specification on applications and enables the identification of the most suitable TO specification to support the implementation of a given application. Furthermore, the paper presents a simple methodology that allows us to fit existing unspecified (or specified with an ad-hoc formalism) TO implementations into the classification framework. The methodology is then applied to analyze the implementation of fixed sequencer-based TO primitives [15] in a particular context, namely primary component group communications [14, 24], which have proven to be an effective paradigm for implementing fault-tolerant distributed applications. In particular, we first introduce a reference architecture based on the virtual synchrony programming model that abstracts group toolkit's implementation details. Then we analyze eight variations of fixed sequencer-based protocols by formally proving their enforced specifications. Let us note that, despite the popularity of these protocols, they have been rarely formalized and/or proved to enforce a precise TO specification. Interested readers can find a similar analysis dealing with privilege-based protocols in [3].

The remainder of this paper is organized as follows. Section 2 presents the system model. Section 3 reports a detailed study of the properties defining the TO problem. Section 4 (i) introduces a hierarchy of TO specifications, (ii) examines differences among the behaviors of correct and faulty processes admitted by each specification (Appendix A presents the corresponding formal analysis), and (iii) presents a simple methodology to classify TO implementations. Section 5 introduces group communication systems, and Section 6 analyzes TO implementations based on a fixed sequencer, formally identifying their enforced specifications. Finally, Section 7 discusses the related work, while Section 8 concludes the paper.

[1] In this case it is likely to face the paradox that a practitioner exploits a TO implementation to preserve the correctness of its application and to increase availability despite failures, whereas this implementation causes a violation of the application's safety exactly upon the occurrence of such an event.

2 System model

The specifications and properties that we introduce in the following sections are based on the asynchronous distributed system model described below. Expressing properties assuming such a system model allows us to apply them to a wide class of synchronous, asynchronous, and partially synchronous real distributed systems in which a TO implementation is developed. We do not consider systems admitting malicious behaviors of processes and/or channels.

Asynchronous distributed system. We consider a system composed by a finite set of processes $\Pi = \{p_1 \dots p_n\}$ communicating by message passing. Each process is equipped with an instance of a *Total Order (TO) Service*. This service interacts with a process through two primitives: TOcast(<message>) (from the process to the TOservice) and TOdeliver(<message>) (from the TO service to the process). To broadcast a message m a process invokes the TOcast(m) primitive. Upon receiving a message m from the underlying communication layer, the TO service waits until m can be delivered to the process (following the specification of the service). Then the TO service invokes TOdeliver(m) which delivers m to the process.

Each process behaves according to its specification until it possibly crashes. A process that never crashes is *correct*, while a process that eventually crashes is *faulty*. The system is asynchronous, i.e. there is no bound (known or unknown) on message transfer delays and on processes' relative speed. For simplicity we assume a message m is broadcast only once.

Histories and runs. Each process $p \in \Pi$ can experience the occurrence of three kinds of events, namely $TOcast(m)$ (produced by the execution of the primitive TOcast(m)), $TOdeliver(m)$ (produced by the execution of the primitive TOdeliver(m)) and $crash$. A history h_p is the sequence of events occurred at p during its lifetime. We denote as $e_i \in h_p$ the i-th event in the history of p. Note that $crash$ may only occur as the last event in the history of a faulty process. A *system run* is a set of histories h_{p_i}, one for each process $p_i \in \Pi$. We denote $\mathcal{R}$ as the set of all possible runs in the system. To characterize runs in $\mathcal{R}$, we introduce the predicates defined in Table 1.

$correct(p)$	$\triangleq$	$crash \notin h_p$
$faulty(p)$	$\triangleq$	$crash \in h_p$
$tocast(p, m)$	$\triangleq$	$TOcast(m) \in h_p$
$todel(p, m)$	$\triangleq$	$TOdeliver(m) \in h_p$
$todel(p, m) < todel(p, m')$	$\triangleq$	$\exists i, j \; e_i = TOdeliver(m) \in h_p \wedge e_j = TOdeliver(m') \in h_p \wedge i < j$

Table 1: Shorthand predicates

Properties and specifications. A *property* P on $\mathcal{R}$ is a first order logic predicate based on the shorthand predicates given in Table 1, and thus defining a set of runs $R_P \subseteq \mathcal{R}$ associated to P. More precisely, a run $r \in \mathcal{R}$ is *admitted* by P, i.e. $r \in R_P$, *iff* r satisfies P. Let P and P' be two properties on $\mathcal{R}$. We say that P is stronger than P' (and that P' is weaker than P), denoted $P \rightarrow P'$, *iff* $R_P \subset R_{P'}$. Note that $P \rightarrow P'$ *iff* $P \Rightarrow P' \wedge \neg(P' \Rightarrow P)$, where $\Rightarrow$ is the logical implication.

A *specification* is a particular kind of property. In particular a *specification* $S(P_1 \dots P_m)$ (with $m \geq 1$) on $\mathcal{R}$ is a predicate on $\mathcal{R}$ composed by the conjunction of m properties, i.e. $S = \bigwedge_{i=1 \dots m} P_i$, defining a set of runs $R_S = \bigcap_{i=1,\dots m} R_{P_i} \subseteq \mathcal{R}$. R_S is composed by all system runs satisfying S. Therefore a specification is a particular type of property, and thus a run $r \in \mathcal{R}$ is *admitted* by a specification $S(P_1 \dots P_m)$, i.e. $r \in R_S = \bigcap_{i=1,\dots m} R_{P_i}$, *iff* r satisfies S.

Given two specifications $S(P_1 \dots P_m)$ and $S'(P'_1 \dots P'_\ell)$, S is stronger than S', denoted $S \rightarrow S'$, *iff* $R_S \subset R_{S'}$. In this case we also say that S' is weaker than S. Note that the $\rightarrow$ relation is transitive, i.e. if $S_1 \rightarrow S_2 \wedge S_2 \rightarrow S_3$ then $S_1 \rightarrow S_3$. Finally, two specifications S and S' are equivalent, denoted $S \equiv S'$, *iff* $R_S = R_{S'}$.

3 Total order properties

To the best of our knowledge, the first coherent description of TO specifications is due to Hadzilacos and Toueg, dating back to 1993 [19]. Since then, total order broadcast has been usually specified by means of four properties, namely *Validity*, *Integrity*, *Agreement*, and *Order*. Informally speaking, a *Validity* property guarantees that (correct) processes eventually deliver the messages they have sent; an *Integrity* property guarantees that no spurious or duplicate messages are delivered; an *Agreement* property ensures that (at least correct) processes deliver the same set of messages; an *Order* property constrains (at least correct) processes delivering the same messages to deliver them in the same order. Each property can be formally defined in distinct ways, thus generating distinct specifications. A typical example of differing formulations of a property of a TO specification is given by its *uniform* and *non-uniform* versions. A uniform property imposes some restrictions on the histories of (at least) correct processes on the basis of some events occurred in the histories of some processes (correct or not). In contrast, a non-uniform property imposes some restrictions on the histories of correct processes on the basis of some events occurred in the histories of some *correct* processes. As a consequence, given a property P, its uniform formulation UP turns out to be stronger than its non-uniform formulation NUP. Therefore, $UP \rightarrow NUP$, and $R_{UP} \subset R_{NUP}$. It is worth noting that uniform properties are meaningful only in certain environments. For instance, uniform properties are not enforceable assuming malicious fault models.

In the following sections we introduce and discuss several formulations of the properties, giving their definitions and highlighting differences in their admitted system runs. For each property we give both the formal definition (see Table 2) and a description in natural language. Concerning the latter, we say that a process $p \in \Pi$ *tocasts* a message m *iff* $tocast(p, m)$, i.e. *iff* $TOcast(m) \in h_p$. Analogously, we say that a process $p \in \Pi$ *todelivers* a message m *iff* $todel(p, m)$, i.e. *iff* $TOdeliver(m) \in h_p$ (see Table 1).

3.1 Validity and Integrity

It is not possible to guarantee that a faulty process eventually delivers messages it sent. Therefore, we only consider the non-uniform version of the *Validity* property, defined as follows:

Non-uniform Validity (NUV). If a *correct* process tocasts a message m, then it eventually todelivers m.

Concerning the *Integrity* property, assuming a crash fault model enables the enforcement of its uniform version without additional overhead. Hence, we only deal with *Uniform Integrity*, defined as follows:

Uniform Integrity (UI). For any message m, every process p todelivers m at most once, and only if m is tocast by some process.

VALIDITY AND INTEGRITY PROPERTIES		
NUV	$\triangleq$	$\forall p \forall m\ tocast(p,m) \wedge correct(p) \Rightarrow todel(p,m)$
UI	$\triangleq$	$\forall m \forall p\ e_i = TOdeliver(m) \in h_p \Rightarrow (\exists q\ tocast(q,m) \wedge \forall e_j \in h_p\ e_i = e_j \Leftrightarrow i = j)$
AGREEMENT PROPERTIES		
UA	$\triangleq$	$\forall p \forall m\ todel(p,m) \Rightarrow (\forall q\ correct(q) \Rightarrow todel(q,m))$
NUA	$\triangleq$	$\forall p \forall m\ correct(p) \wedge todel(p,m) \Rightarrow (\forall q\ correct(q) \Rightarrow todel(q,m))$
ORDER PROPERTIES		
$SUTO$	$\triangleq$	$\forall p \forall m, m'\ todel(p,m) < todel(p,m') \Rightarrow (\forall q\ todel(q,m') \Rightarrow todel(q,m) < todel(q,m'))$
$SNUTO$	$\triangleq$	$\forall p \forall m, m'\ correct(p) \wedge todel(p,m) < todel(p,m') \Rightarrow (\forall q\ correct(q) \wedge todel(q,m') \Rightarrow todel(q,m) < todel(q,m'))$
$WUTO$	$\triangleq$	$\forall p, q \forall m, m'\ todel(p,m) \wedge todel(p,m') \wedge todel(q,m) \wedge todel(q,m') \Rightarrow (todel(p,m) < todel(p,m') \Leftrightarrow todel(q,m) < todel(q,m'))$
$WNUTO$	$\triangleq$	$\forall p, q \forall m, m'\ correct(p) \wedge correct(q) \wedge todel(p,m) \wedge todel(p,m') \wedge todel(q,m) \wedge todel(q,m') \Rightarrow (todel(p,m) < todel(p,m') \Leftrightarrow todel(q,m) < todel(q,m'))$

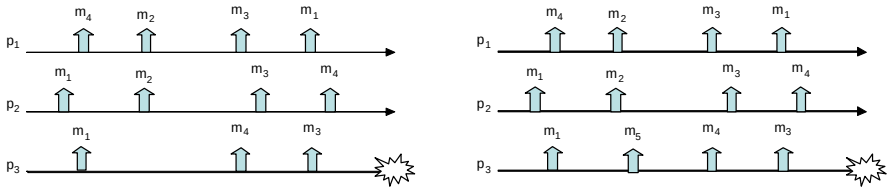
Table 2: Formal definition of the properties defining TO specifications

3.2 The Agreement property

An *Agreement* property imposes some constraints on the sets of messages delivered by (at least correct) processes. This property is usually specified according to one of the following formulations:

Uniform Agreement (UA). If a process todelivers a message m, then all correct processes eventually todeliver m;

Non-uniform Agreement (NUA). If a *correct* process todelivers a message m, then all correct processes eventually todeliver m.



(a) A run in R_{UA}

(b) A run in $R_{NUA} - R_{UA}$

Figure 1: Differences between UA and NUA

Both formulations constrain all correct processes to deliver the same set of messages. The difference between the two properties lies in the restrictions imposed on the set of messages delivered by faulty processes. Let us first consider UA. This property imposes that each message delivered by some process (correct or not) is also delivered by each correct process. In contrast, faulty processes are allowed to skip[2] the delivery of some messages delivered by some other

[2]In this paper we use the term *skip* to denote the fact that a process p misses to deliver a message m which is instead delivered by some other process. This can occur, for example, because the sender of m crashes after m has reached only a subset of its destinations, not including p.

process. This implies that in any run satisfying UA each faulty process delivers a subset of the messages delivered by correct processes. Figure 1(a) depicts a run satisfying UA.[3]

Differently from UA, NUA admits faulty processes to deliver messages that are not delivered by any other process, e.g. message m_5 delivered by p_3 in Figure 1(b). Therefore, in any run satisfying NUA, the set of messages delivered by each faulty process only intersects the set of messages delivered by correct processes, being the intersection possibly empty.

Let us note that a communication primitive satisfying an *Agreement* property along with NUV and UI is usually called *reliable broadcast* [19].

3.3 The Order property

All formulations of the *Order* property force correct processes to deliver messages in the same order. However, they differ (i) in the restrictions imposed on deliveries made by faulty processes, and (ii) in the way in which the order of message deliveries is defined. In particular, the latter differentiation gives rise to *strong* and *weak Order* properties.

Strong vs. weak *Order* properties. A weak *Order* property defines a total order by requiring the same order of delivery for each *pair* of messages delivered by two distinct processes. This restriction does not prevent a process p to skip the delivery of some message. Therefore, it permits the occurrence of gaps in the sequence of messages delivered by p with respect to those delivered by other processes. In contrast, a strong *Order* property avoids gaps in the sequence of delivered messages as it requires that two processes delivering a message m have delivered exactly the same ordered sequence of messages up to m.

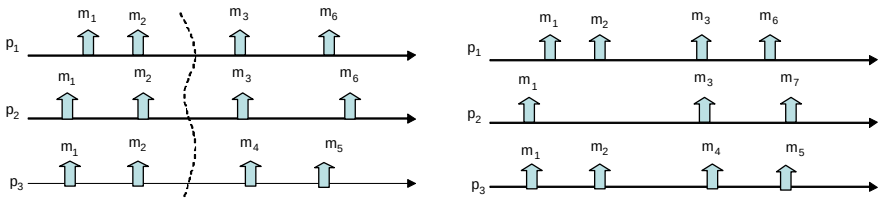
Combining uniform and non-uniform with strong and weak formulations, we obtain four *Order* properties, namely Strong Uniform Total Order ($SUTO$), Strong Non-uniform Total Order ($SNUTO$), Weak Uniform Total Order ($WUTO$) and Weak Non-uniform Total Order ($WNUTO$), reported in Table 2. The remainder of this section analyzes differences and implications among these four formulations.

Strong Uniform Total Order ($SUTO$). If some process todelivers some message m before message m', then a process todelivers m' only after it has todelivered m.

$SUTO$ imposes that if some processes deliver the same messages, they all deliver these messages in the same order. However, if any two processes deliver distinct messages as the ith delivered message, they are forced to successively deliver only distinct messages. This is the case of processes p_1, p_2, and p_3 of Figure 2(a), in which before the dashed line the sets of delivered messages coincide, and thus their ordering. Then p_1 and p_2 deliver m_3 that is skipped by p_3. As a consequence, the following messages delivered by p_3 must be disjoint from those delivered by p_1 and p_2.

Weak Uniform Total Order ($WUTO$). If processes p and q both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'.

[3]Since *Agreement* and *Order* are the only properties for which uniform and non-uniform formulations are being considered, we can focus only on message deliveries and process crashes as the events characterizing system runs. As a consequence, the figures of this section contain only these events.



(a) A run in R_{SUTO} (b) A run in $R_{WUTO} - R_{SUTO}$

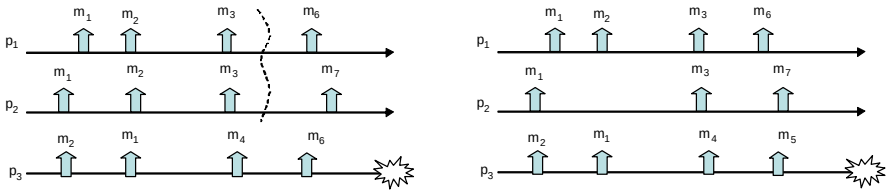
Figure 2: Differences between $SUTO$ and $WUTO$

The following Lemma shows that $WUTO$ is weaker than $SUTO$, i.e. $SUTO \rightarrow WUTO$.

Lemma 1. $SUTO \rightarrow WUTO$

Proof. Assume by contradiction that $R_{SUTO} - R_{WUTO} \neq \emptyset$ and let r be a run such that $r \in R_{SUTO} \wedge r \notin R_{WUTO}$. In order to violate $WUTO$, there must exist two processes p, q and two messages m, m' in r such that p delivers m before m' while q delivers m' before m. However, this violates $SUTO$, which is a contradiction. As a consequence $R_{SUTO} \subseteq R_{WUTO}$. Furthermore, Figure 2(b) depicts a run in $R_{WUTO} - R_{SUTO}$. Therefore $R_{SUTO} \subset R_{WUTO}$, i.e. $SUTO \rightarrow WUTO$. □

Being a weak *Order* property, $WUTO$ allows the occurrence of gaps in the sequence of messages delivered by distinct processes. As an example, Figure 2(b) depicts a run in $R_{WUTO} - R_{SUTO}$ in which p_2 delivers m_3 after having skipped m_2, while p_1 delivers both m_2 and m_3 in this order.



(a) A run in $R_{SNUTO} - R_{SUTO}$ (b) A run in $R_{WNUTO} - R_{WUTO}$

Figure 3: Differences between $SNUTO$ and $SUTO$ and between $WNUTO$ and $WUTO$

Strong Non-uniform Total Order ($SNUTO$). If some *correct* process todelivers some message m before message m', then a *correct* process todelivers m' only after it has todelivered m.

$SNUTO$ is the non-uniform counterpart of $SUTO$. Hence, due to the relation between uniform and non-uniform properties, $SUTO \rightarrow SNUTO$. The difference between these two properties lies in the behavior of faulty processes. In fact, $SNUTO$ allows faulty processes to

deliver an arbitrary set of messages and to deliver them in an arbitrary order (e.g. process p_3 in Figure 3(a)). In contrast, correct processes are forced to agree on a prefix of the ordered set of delivered messages. Furthermore, any two correct processes delivering distinct messages as the ith delivered message has to successively deliver only disjoint set of messages (see processes p_1 and p_2 in Figure 3(a)).

Weak Non-uniform Total Order ($WNUTO$). If *correct* processes p and q both todeliver messages m and m', then p todelivers m before m' if and only if q todelivers m before m'.

$WNUTO$ is the non-uniform counterpart of $WUTO$. Hence $WUTO \rightarrow WNUTO$. Furthermore, in a way similar to Lemma 1, it is possible to show that $SNUTO \rightarrow WNUTO$. As $SNUTO$, $WNUTO$ allows faulty processes to deliver messages in arbitrary order (see Figure 3(b)). Moreover, processes (correct or not) are free to deliver distinct sets of messages, i.e. $WNUTO$ allows the occurrence of gaps in the sequence of message deliveries.

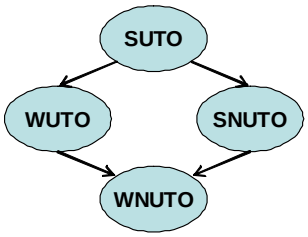


Figure 4: Relations among *Order* properties

Figure 4 summarizes the previous discussion by reporting the relations shown to hold among *Order* properties. Note that, from the transitivity of the $\rightarrow$ relation, immediately follows that $SUTO \rightarrow WNUTO$.

4 A hierarchy of total order specifications

Assuming *Non-Uniform Validity* and *Uniform Integrity*, it is possible to combine *Agreement* and *Order* properties to obtain eight TO specifications. We denote $TO(A, O)$ the TO specification $S(NUV, UI, A, O)$, where $A \in \{UA, NUA\}$ and $O \in \{SUTO, SNUTO, WUTO, WNUTO\}$.

Lemma 2 and Corollary 1 below prove that $TO(NUA, SNUTO) \equiv TO(NUA, WNUTO)$ and that $TO(UA, SNUTO) \equiv TO(UA, WNUTO)$.

Lemma 2. $TO(NUA, SNUTO) \equiv TO(NUA, WNUTO)$

Proof.

1. ($\subseteq$) By contradiction. Let r be a run such that $r \in R_{SNUTO} \cap R_{NUA} \cap R_{NUV} \cap R_{UI} \wedge r \notin R_{WNUTO} \cap R_{NUA} \cap R_{NUV} \cap R_{UI}$. From $r \in R_{SNUTO} \cap R_{NUA} \cap R_{NUV} \cap R_{UI}$, it follows that r must satisfy $NUA \wedge NUV \wedge UI$. As a consequence r violates $WNUTO$, i.e. there exists two correct processes p, q and two messages m, m' such that p delivers m before m' in r while q delivers m' before m in r. However, this contradicts the hypothesis that r satisfies $SNUTO$.

2. ($\supseteq$) By contradiction. Let r be a run such that $r \in R_{WNUTO} \cap R_{NUA} \cap R_{NUV} \cap R_{UI} \wedge r \notin R_{SNUTO} \cap R_{NUA} \cap R_{NUV} \cap R_{UI}$. From $r \in R_{WNUTO} \cap R_{NUA} \cap R_{NUV} \cap R_{UI}$, it follows that r must satisfy $NUA \wedge NUV \wedge UI$. As a consequence, it violates $SNUTO$. Suppose that p, q are two correct processes and m, m' are two messages in r, and that p delivers m before m'. Since r violates $SNUTO$, we have the following two cases:

   (a) q delivers m' without delivering m. This contradicts the hypothesis that r satisfies NUA.

   (b) q delivers m after delivering m'. This contradicts the hypothesis that r satisfies $WNUTO$. □

Corollary 1. $TO(UA, SNUTO) \equiv TO(UA, WNUTO)$

Proof.

1. ($\subseteq$) Let r be a run such that $r \in R_{SNUTO} \cap R_{UA} \cap R_{NUV} \cap R_{UI}$. Since $UA \rightarrow NUA$, $r \in R_{UA} \Rightarrow r \in R_{NUA}$. Therefore, $r \in R_{SNUTO} \cap R_{NUA} \cap R_{NUV} \cap R_{UI}$. From Lemma 2, it follows that $r \in R_{WNUTO} \cap R_{NUA} \cap R_{NUV} \cap R_{UI}$. As a consequence $r \in R_{WNUTO}$, and thus $r \in R_{WNUTO} \cap R_{UA} \cap R_{NUV} \cap R_{UI}$.

2. ($\supseteq$) Let r be a run such that $r \in R_{WNUTO} \cap R_{UA} \cap R_{NUV} \cap R_{UI}$. Since $UA \rightarrow NUA$, $r \in R_{UA} \Rightarrow r \in R_{NUA}$. Therefore, $r \in R_{WNUTO} \cap R_{NUA} \cap R_{NUV} \cap R_{UI}$. From Lemma 2, it follows that $r \in R_{SNUTO} \cap R_{NUA} \cap R_{NUV} \cap R_{UI}$. As a consequence $r \in R_{SNUTO}$, and thus $r \in R_{SNUTO} \cap R_{UA} \cap R_{NUV} \cap R_{UI}$. □

These results can be intuitively explained noting that $WNUTO$ and $SNUTO$ differ from the restrictions they impose on the set of messages delivered by correct processes and not from the order of deliveries. As a consequence, they give rise to equivalent specifications when combined with an *Agreement* property, which forces correct processes to deliver the same set of messages. Therefore we can restrict our attention to six (out of the expected eight) significant TO specifications, considering $O \in \{SUTO, WUTO, WNUTO\}$.

It is possible to exploit the precedence relations derived for *Order* and *Agreement* properties to identify relations among TO specifications. To this aim, it is first worth noting that given two specifications differing only for the definition of the *Agreement* property, i.e. $TO(A, O)$ and $TO(A', O)$, then $TO(A, O) \rightarrow TO(A', O)$ *iff* $A \rightarrow A'$ and $R_{TO(A',O)} - R_{TO(A,O)} \neq \emptyset$. In this case any run $r \in R_{TO(A',O)} - R_{TO(A,O)}$ satisfies $\neg A$, i.e. $\neg A$ characterizes all runs in $R_{TO(A',O)} - R_{TO(A,O)}$. The same reasoning applies to specifications differing only for their *Order* property. As an example, observing that $UA \rightarrow NUA$ and that the run depicted in Figure 6(b) belongs to $R_{TO(NUA,SUTO)} - R_{TO(UA,SUTO)}$, it follows that $TO(UA, SUTO) \rightarrow TO(NUA, SUTO)$. Further relations among specifications are due to the transitivity of the $\rightarrow$ relation. For example, $TO(UA, SUTO) \rightarrow TO(NUA, WNUTO)$.

Figure 5 depicts the transitive reduction of the $\rightarrow$ relation among TO specifications. This reduction represents a hierarchy of specifications. Note that any two specifications S and S' connected by an edge in the hierarchy differ only for distinct formulations of a single property, say P and P', such that $P \rightarrow P'$. The edge is then labelled with the predicate $\neg P$ (see Table 3 for its definition) that can be used to characterize runs belonging to $R_{S'} - R_S$.

Let us note that the root of the hierarchy, i.e. $TO(UA, SUTO)$, is the specification closest to the intuitive notion of total order broadcast, as it imposes that *the set of messages delivered*

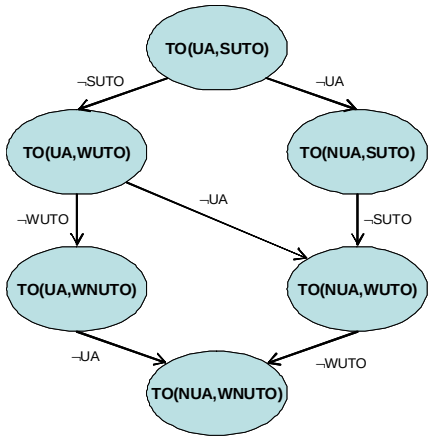


Figure 5: A hierarchy of TO specifications

$\neg UA$	$\triangleq$	$\exists p, q \exists m\ todel(p,m) \wedge \neg todel(q,m) \wedge correct(q)$
$\neg SUTO$	$\triangleq$	$\exists p, q \exists m, m'\ todel(p,m) < todel(p,m') \wedge todel(q,m') \wedge (\neg todel(q,m) \vee todel(q,m') < todel(q,m))$
$\neg WUTO$	$\triangleq$	$\exists p, q \exists m, m'\ todel(p,m) < todel(p,m') \wedge todel(q,m') < todel(q,m)$

Table 3: Predicates associated with direct relations in the hierarchy

by each process is a prefix of the ordered set of messages that is delivered by all correct processes. This is due to the interaction between $SUTO$ and UA. To explain this, let us consider a faulty process skipping the delivery of a message delivered by correct processes, e.g. p_3 omitting m_3 in Figure 6(a). Due to $SUTO$, the messages delivered by p_3 after skipping m_3 have not to be delivered by processes that deliver m_3. Now suppose that p_3 delivers a new message m *after* having skipped m_3 and *before* crashing. To let the run satisfy UA, m has to be delivered by correct processes. However, in that case the run would violate $SUTO$. As a consequence p_3 may not deliver *any* message after skipping the delivery of a message that is delivered by some other process. This is sufficient to ensure that all processes deliver all messages in the same order before crashing. In contrast, weaker specifications admit runs in which faulty processes are allowed to exhibit a wider set of behaviors, as discussed in the following section. It is worth noting that weaker specifications are those implemented by several real systems, e.g. Ensemble [20], JavaGroups [6].

4.1 On the behavior of faulty processes

Each TO specification constrains all correct processes to deliver exactly the same ordered set of messages. On the contrary, each specification poses its own restrictions on the ordered set of messages that can be delivered by each faulty process. Differences among the sequences of delivered messages can be characterized using the following execution patterns.

EP1: a faulty process p delivers a prefix of the ordered set of messages delivered by correct processes;

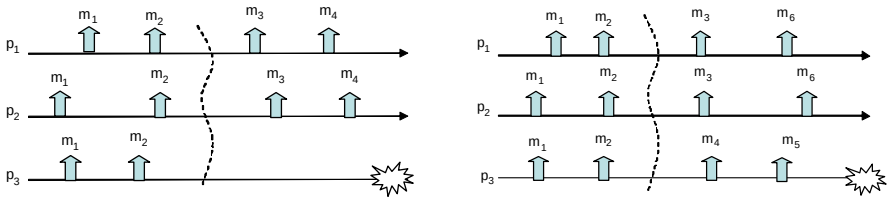
EP2: a faulty process p delivers some messages that are not delivered by correct

processes;

EP3: a faulty process p skips the delivery of some message delivered by correct processes;

EP4: a faulty process p delivers some message in an order different from correct processes.

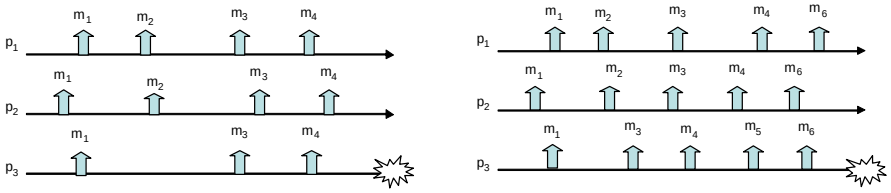
Each specification permits the occurrence of one or more of the above execution patterns. Moreover, from the definition of the $\rightarrow$ relation, it follows that for each pair of specifications $S, S' : S \rightarrow S'$, S' allows at least all execution patterns admitted by S. For example, $TO(UA, SUTO)$ allows EP1 while $TO(UA, WUTO)$ allows EP1 and EP3. Table 4 shows for each specification the admitted execution patterns. Let us note that these execution patterns are formally derived from specifications (see Appendix A).



(a) A run in $R_{TO(UA,SUTO)}$

(b) A run in $R_{TO(NUA,SUTO)} - R_{TO(UA,SUTO)}$

Figure 6: Differences between $TO(UA, SUTO)$ and $TO(NUA, SUTO)$



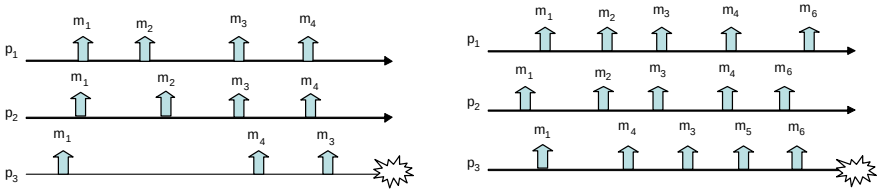
(a) A run in $R_{TO(UA,WUTO)} - R_{TO(UA,SUTO)}$

(b) A run in $R_{TO(NUA,WUTO)} - R_{TO(UA,WUTO)}$

Figure 7: Differences between $TO(UA, WUTO)$ and $TO(NUA, WUTO)$

4.2 Associating implementations to specifications

When dealing with a TO implementation with a complex or loose specification (e.g. [9, 21]) it could be difficult to identify its guarantees. This problem can be solved by classifying the TO implementation with respect to the hierarchy of Figure 5 and then inspecting Table 4 to identify the admitted behavior for faulty processes. To this end, let us first introduce the following definition.



(a) A run in $R_{TO(UA,WNUTO)} - R_{TO(UA,WUTO)}$

(b) A run in $R_{TO(NUA,WNUTO)} - R_{TO(UA,WNUTO)}$

Figure 8: Differences between $TO(UA, WNUTO)$ and $TO(NUA, WNUTO)$

TO specification	Admitted differences in the histories of processes p (faulty) and q (correct)	Example
$TO(UA, SUTO)$	EP1	Figure 6(a)
$TO(UA, WUTO)$	EP1 - EP3	Figure 7(a)
$TO(UA, WNUTO)$	EP1 - EP3 - EP4	Figure 8(a)
$TO(NUA, SUTO)$	EP1 - EP2	Figure 6(b)
$TO(NUA, WUTO)$	EP1 - EP2 - EP3	Figure 7(b)
$TO(NUA, WNUTO)$	EP1 - EP2 - EP3 - EP4	Figure 8(b)

Table 4: Possible differences between the behavior of faulty and correct processes

Definition. Let $\mathcal{I}$ be a TO implementation and let $R_{\mathcal{I}}$ be the set of runs that $\mathcal{I}$ can generate. $\mathcal{I}$ enforces a TO specification S *iff*:

1. $R_{\mathcal{I}} \subseteq R_S$, and
2. $\forall S'\ S' \rightarrow S \Rightarrow R_{\mathcal{I}} \nsubseteq R_{S'}$.

Therefore the problem of finding the TO specification enforced by a TO implementation $\mathcal{I}$ boils down to find a TO specification S defining the smallest superset of $R_{\mathcal{I}}$. It is worth noting that, given two specifications S and S' connected by an edge labelled with $\neg P$ in the hierarchy of Figure 5, a sufficient condition to state that $\mathcal{I}$ does not implement S but possibly implements S' is that there exists a run in $R_{\mathcal{I}}$ that satisfies $\neg P$. As an example, consider $TO(UA, SUTO) \rightarrow TO(NUA, SUTO)$. Verifying that a run $r \in R_{\mathcal{I}}$ satisfies $\neg UA$, which is the label of the edge between these two specifications in Figure 5, allows us to state that $R_{\mathcal{I}} \nsubseteq R_{TO(UA,SUTO)}$, but possibly $R_{\mathcal{I}} \subseteq R_{TO(NUA,SUTO)}$.

A methodology to find the implemented specification S is to navigate the hierarchy according to the following rules:

1. verify that $\mathcal{I}$ is a TO implementation, i.e. $R_{\mathcal{I}} \subseteq R_{TO(NUA,WNUTO)}$. If affirmative, go to step 2, otherwise $\mathcal{I}$ is not a TO implementation;

2. set the current specification S to the one on the top of the hierarchy, i.e. $TO(UA, SUTO)$;

3. while there exists an unchecked specification S' such that $S \rightarrow S'$ belongs to the transitive reduction of $\rightarrow$ depicted in Figure 5, check if there exists a run $r \in R_{\mathcal{I}}$ satisfying the

associated predicate $\neg P$. If affirmative, repeat this step setting current the specification S to S'. Otherwise go to step 4;

4. S represents the specification implemented by $\mathcal{I}$.

5 Primary component group communication systems

In this section we briefly introduce the main features of primary component group communication systems, which are by far one of the most successful class of systems implementing TO primitives.

5.1 Reference architecture and the virtual synchrony programming model

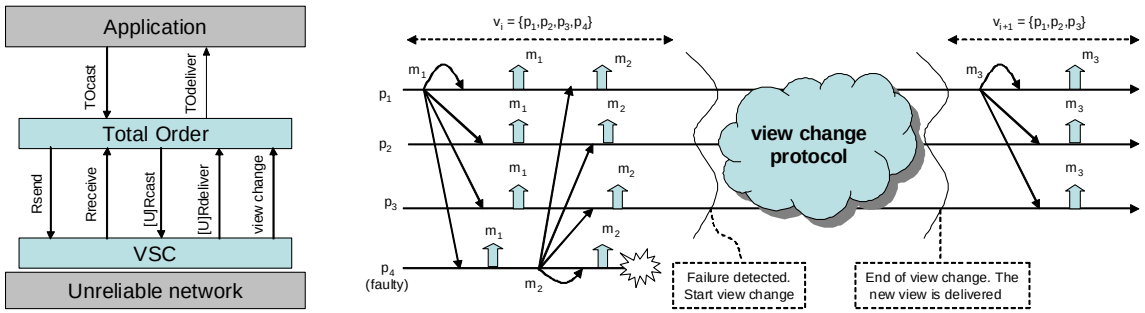
Group communication systems adopt several distinct architectures [23]. For the sake of clarity, in the remainder of this paper we use a simplified architecture depicted in Figure 9(a), in which a Total Order layer implements a TO specification by relying on another layer, namely VSC, which provides virtually synchronous communications [8].[4] According to the virtual synchrony programming model, processes are organized into groups. Groups are *dynamic*, i.e. processes are allowed to join and voluntarily leave a group using appropriate primitives. Furthermore, faulty processes are excluded by groups after crashing. A *group membership service* provides each process of a group with a consistent *view* v_i composed by the identifiers of all non-crashed processes currently belonging to the group. Upon a membership change, processes agree on a new view through a *view change protocol*. At the end of this protocol, group members are provided with a view v_{i+1} (i) that is delivered to all the members of v_{i+1} through a view change event, and (ii) contains the identifier of all the members that deliver v_{i+1} (see Figure 9(b)). We assume a *primary component* membership service, e.g. [10], guaranteeing that all members of the same group observe the same sequence of views as long as they stay in the group. In this context, the VSC layer guarantees (i) that membership changes of a group occur in the same order in all the members that stay within the group, and (ii) that membership changes are totally ordered with respect to all messages sent by members. It is worth noting that the primary component membership service is not implementable in a non-blocking manner in asynchronous systems [12].[5]

To formalize these concepts, we let the VSC layer export primitives of the membership service, namely *view_change*, as well as a set of communication primitives, namely $Rsend(m)$ and $Rreceive(m)$ (for reliable point-to-point communications), $Rcast(m)$ and $Rdeliver(m)$ (for non-uniform reliable broadcast communications), and $URcast(m)$ and $URdeliver(m)$ (for uniform reliable broadcast communications). In order to parameterize TO protocols with respect to the employed reliable broadcast primitive, we use $[U]Rcast(m)$ and $[U]Rdeliver(m)$ to denote the invocation of *any* reliable broadcast primitive, uniform or not.

These primitives satisfy the following properties:

[4] Let us remark that other approaches incorporating the implementation of *Order* and *Agreement* properties into a single protocol are possible, e.g. [13].

[5] In *partitionable* systems, groups may partition into subgroups (or *components*), e.g. due to network failures, and members of distinct subgroups can deliver distinct sequences of views. In this setting, specifying a total order primitive can drive to complex specifications, e.g. [14], whose usefulness has still to be verified [25]. However, non-blocking implementations of partitionable group communication systems are feasible in asynchronous systems.



(a) Architecture

(b) The Virtual Synchrony property

Figure 9: Group communication systems

VSC1. If a correct process p Rsends a message m to a correct process q, then q eventually Rreceives m.

VSC2. For each message m, a process p Rreceives m at most once, and only if m was Rsent to p by some process q.

These properties basically state that point-to-point communications among processes occur through quasi-reliable channels [7].

Concerning $[U]Rcast(m)$ and $[U]Rdeliver(m)$, these primitives satisfy the following properties [14, 27]:

VSC3. If a correct process p [U]Rcasts a message m, then it eventually [U]Rdelivers m.

VSC4. For each message m, each process [U]Rdelivers m at most once, and only if m was [U]Rcast by some process.

VSC5. If a process (respectively, a correct process) p URdelivers (respectively, Rdelivers) m in view v, then all processes which are either correct or deliver a view change event in v URdeliver (resp. Rdeliver) m.

VSC6. If a process p [U]Rdelivers a message m in view v and a process q [U]Rcasts m in view v' then $v = v'$.

These properties guarantee that processes agree on the set of messages delivered before installing a new view, thus enforcing virtual synchrony. Furthermore, property VSC6 (named also *Sending View Delivery* [14]), guarantees that messages are delivered in the view they were sent.

5.2 Static vs. dynamic group communications.

Following the model proposed by Hadzilacos and Toueg in [19], the properties introduced in Section 3 are based on a system model that does not take process joins into account. In contrast, group communication systems typically support dynamic groups, which lead to more complex

specifications, e.g. [14, 25]. We now show how the TO specifications introduced in Section 4 can be used to classify also dynamic TO implementations. For this purpose, we introduce the notion of *static sub-run*, i.e. a portion of the overall computation of a system supporting dynamic groups in which *join events may only appear at the beginning of the sub-run*. A given run of any group communication system supporting dynamic groups can thus be decomposed in its static sub-runs, i.e. sub-computations of an overall run starting with a view change event with at least a joining process and ending before the join of another process (if any). Considering the computation depicted in Figure 10, this run can be decomposed in three static sub-runs, namely sr_1, sr_2, sr_3. It is important to note that a sub-run can be described with events and process histories as those introduced in Section 2, i.e. $TOcast(m)$, $TOdeliver(m)$, and $crash$. For example the sub-run sr_2 depicted in Figure 10 is composed by the histories of processes p_1, p_2 and p_4 containing the message delivery events of m_4 and m_5. Moreover, p_1 is correct in sr_1 and sr_2 while it is faulty in sr_3.

In this dynamic context, the answer to the question *"which TO specification is enforced by an implementation $\mathcal{I}$?"*, can be given by considering the set of all static sub-runs (instead of the set of runs) that $\mathcal{I}$ can produce as the set $R_{\mathcal{I}}$ used in the methodology described in Section 4.2.

As an example, in the run depicted in Figure 10, sub-runs sr_1 and sr_2 satisfy $TO(UA, SUTO)$, while sr_3 only satisfies $TO(NUA, SUTO)$ (due to p_5 delivering message m_9). Therefore, an implementation $\mathcal{I}$ that may generate this run enforces at most $TO(NUA, SUTO)$ (i.e. $\mathcal{I}$ does not enforce $TO(UA, SUTO)$).

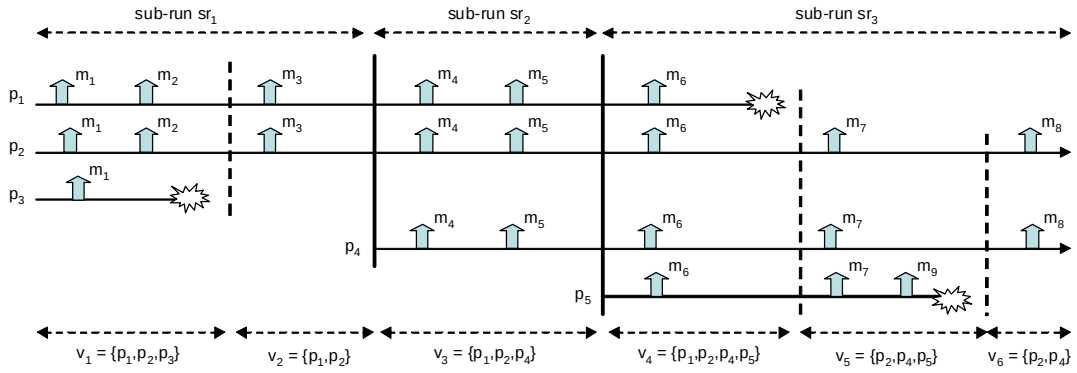


Figure 10: A run of a system supporting dynamic process joins

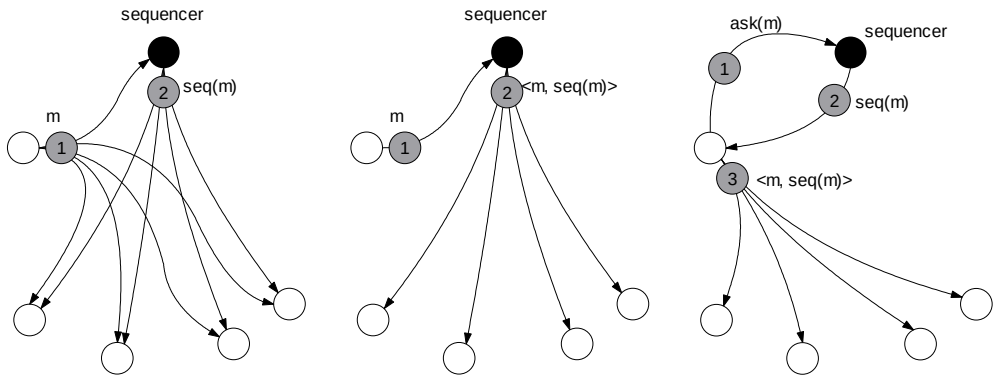
6 Fixed sequencer-based TO implementations in group communication systems

The most widely used protocols implementing the Total Order layer can be classified in *fixed sequencer* and *privilege-based* [16]. Interested readers are referred to [16] for a description of several other classes of TO implementations.

In the following we describe eight variations of fixed sequencer protocols and classify them into the proposed hierarchy of specifications exploiting the companion methodology of Section 4.2. A similar analysis carried out for privilege-based protocols ca be found in [3].

6.1 Fixed sequencer protocols

In fixed sequencer protocols a particular process, i.e. the *sequencer*, is responsible for defining message ordering. This process is elected after each view change, usually on the basis of a deterministic rule applied to the current view, and defines a total order of messages by assigning to each message a unique and consecutive sequencer number. The sequence number assigned to a message is sent to all members, which deliver messages according to these numbers. These steps can be implemented using the following communication patterns[6].



(a) Broadcast-Broadcast (b) Send-Broadcast (c) Ask-Broadcast

Figure 11: Communication pattern of fixed sequencer protocols (borrowed from [16])

- **Broadcast-Broadcast(BB).** The sender broadcasts message m to all members. Upon receiving m, the sequencer assigns a sequence number seq to m, denoted $seq(m)$ and then broadcasts $seq(m)$ to all members (see Figure 11(a)). As an example, the Ensemble system [20] implements this pattern;
- **Send-Broadcast(SB).** The sender sends message m to the sequencer, which assigns a sequence number $seq(m)$ and then broadcasts the pair $\langle m, seq(m)\rangle$ to all members (see Figure 11(b)). This pattern is implemented, for example, by the Ensemble system [20];
- **Ask-Broadcast(AB).** The sender first gets a sequence number to assign to m from the sequencer via a simple rendezvous, then it broadcasts the pair $\langle m, seq(m)\rangle$ to all members (see Figure 11(c)). JavaGroups [6] is an example of a system implementing this pattern.

6.2 A generic fixed sequencer protocol

Figure 12 shows the pseudo-code of a generic fixed sequencer protocol. By instantiating procedures x-Recover(), x-TOcast(), x-Deliver() and x-Receive(), it is possible to implement BB, SB and AB fixed sequencer protocols (see Figures 13, 14 and 15, respectively).

[6]In a companion paper [5], we show how group toolkits implement such communication patterns

```
GENERIC FIXED SEQUENCER PROTOCOL(initial_view)
 1  INTEGER seqnum_p ← 0;
 2  INTEGER last_delivered_p ← 0;
 3  SET received_p ← ∅;
 4  SET received_seqs_p ← ∅;
 5  SET pending_p ← ∅;
 6  VIEW current_view_p ← initial_view;
 7  procedure DELIVER()
 8    while ∃m : m ∈ received_p ∧ ⟨id(m), seq(m)⟩ ∈ received_seqs_p ∧ seq(m) = last_delivered_p + 1 do
 9      TOdeliver(m);
10      last_delivered_p ← last_delivered_p + 1;
11      received_p ← received_p \ {m};
12      received_seqs_p ← received_seqs_p \ {⟨id(m), seq(m)⟩};
13  when (view_change(new_view)) do
14    while received_p ≠ ∅ do
15      MESSAGE m ← Φ(received_p, received_seqs_p);
16      TOdeliver(m);
17      received_p ← received_p \ {m};
18      received_seqs_p ← received_seqs_p \ {⟨id(m), seq(m)⟩};
19    seqnum ← 0;
20    last_delivered_p ← 0;
21    received_seqs_p ← ∅;
22    current_view_p ← new_view;
23    x-RECOVER();
24  procedure TOCAST(m)
25    pending_p ← pending_p ∪ {m};
26    x-TOCAST(m);
27  when ([U]Rdeliver(⟨m, CI⟩)) do
28    if (CI.view = current_view) then
29      x-DELIVER(⟨m, CI⟩)
30  when (Rreceive(⟨m, CI⟩) from q) do
31    if (CI.view = current_view) then
32      x-RECEIVE(⟨m, CI⟩, q)
```

Figure 12: Generic fixed sequencer protocol run by process p

We assume that each message m has a unique identifier, denoted by $id(m)$. We also denote the sequence number (if any) assigned to m by $seq(m)$. At the Total Order layer, messages are exchanged along with a control information (CI). A control information is a tuple $[type, id, seqnum, view]$, where (i) $type$ is a label distinguishing the message type, (ii) id represents the unique message identifier, (iii) $seqnum$ represents a sequence number, and $view$ is the view in which the message is sent. We denote as $CI.type$, $CI.id$, $CI.seqnum$ and $CI.view$ the corresponding fields $type$, id, $seqnum$ and $view$ of CI.

For each process p, the generic fixed sequencer protocol maintains an internal state composed by the following variables:

- $seqnum_p$ (line 1), used by the sequencer process to store the last sequence number assigned to a message.
- $last\_delivered_p$ (line 2), storing the sequence number of the last todelivered message;

- $received_p$ (line 3), storing received messages that are not yet todelivered;
- $received\_seqs_p$ (line 4), storing $\langle message\ identifier, sequence\ number \rangle$ pairs, one for each received sequence number associated to a not yet todelivered message;
- $pending_p$ (line 5), storing messages which have been tocast but not yet received;
- $current\_view_p$ (line 6), storing the last received view.

The protocol embeds an internal procedure, namely DELIVER() (line 7), which is used to todeliver received messages according to sequence numbers. In particular, this procedure repeatedly checks if the next messages to be todelivered have been received (line 8), and in the affirmative it todelivers these messages, updating local variables accordingly (lines 9-12).

The protocol handles several events. Upon the occurrence of the view change event at p (line 13), messages belonging to $received_p$ are todelivered (line 16). The order of deliveries is determined by a deterministic function Φ (line 15) defined as follows.

$$\Phi = \begin{cases} \bot & \text{if } received_p = \emptyset \\ m & \text{if } (m \in received_p \wedge (\langle id(m), seq(m) \rangle \in received\_seqs_p \ \wedge min(seq(m))) \vee \\ & \quad (min(id(m)) \wedge \nexists m'(m' \in received_p \wedge \langle id(m'), seq(m') \rangle \in received\_seqs_p)) \end{cases}$$

where

$$\begin{array}{lcl} min(seq(m)) & \triangleq & \forall m' \in received_p \ \wedge (\langle id(m'), seq(m') \rangle \in received\_seqs_p \ (seq(m) < seq(m') \\ min(id(m)) & \triangleq & \forall m' \in received_p \ id(m) < id(m') \end{array}$$

The next message that must be todelivered is thus the message with the lowest sequence number (if any) or the message with the lowest identifier. After all messages have been todelivered, local variables are reset (lines 19-21) and the X-RECOVER() procedure is called (line 23). This procedure simply retransmits (according to the adopted communication pattern) pending messages which could have been lost due to failures (see Figures 13, 14 and 15).

The invocation of the TOCAST(m) primitive (line 24), the arrival of a broadcast message (line 27) and the arrival of an unicast message (line 30) from the VSC layer are handled in a specific way, depending on the adopted communication pattern. Let us note that, despite the VSC layer guarantees Sending View Delivery (property VSC6), this property does not hold at the Total Order layer due to the asynchrony of the interaction between VSC and Total Order[7]. As a consequence, messages could be received by the Total Order layer in a view different from the one in which they were sent. These messages are filtered out (lines 28 and 31) to avoid them receiving twice (which could occur upon retransmissions).

6.2.1 Broadcast-Broadcast fixed sequencer protocol

The pseudo-code of BB procedures is shown in Figure 13.

[7]Let us denote p^{VSC} and p^{TO} the code executed by the VSC layer and the TO layer at p, respectively. Suppose that p^{TO} is in view v (that is, v is the last view delivered by p^{VSC} to p^{TO}) and a view change occurs at p^{VSC}. Meanwhile, p^{TO} executes `TOcast(`m`)` (which internally calls [U]Rcast(m)) and this happens *after* the view change at p^{VSC} but *before* the new view v' is delivered to p^{TO}. Subsequently, v' is delivered to p^{TO} and `TOdeliver(`m`)` is called. Therefore, from the point of view of p^{TO}, m is sent in view v and it is delivered in view v', which violates Sending View Delivery at p^{TO}, even if the property still holds at p^{VSC}.

```
BB fixed sequencer protocol()
33  procedure BB-TOcast(m)
34    [U]Rcast(⟨m, [ApplMsg, ⊥, ⊥, current_view]⟩);
35  procedure BB-Deliver(⟨m, CI⟩)
36    switch
37      case CI.type = ApplMsg :
38            received_p ← received_p ∪ {m};
39            pending_p ← pending_p \ {m};
40            if (Sequencer?(current_view)) then
41              seqnum ← seqnum + 1;
42              [U]Rcast(⟨⊥, [SeqNum, id(m), seqnum, current_view]⟩);
43      case CI.type = SeqNum :
44            received_seqs_p ← received_seqs_p ∪ {⟨CI.id, CI.seqnum⟩};
45    Deliver();
46  procedure BB-Recover()
47    for each m ∈ pending_p
48      [U]Rcast(⟨m, [ApplMsg, ⊥, ⊥, current_view]⟩);
```

Figure 13: Pseudo-code of the BB fixed sequencer protocol

When BB-TOcast(m) is called, the pair $\langle m, [ApplMsg, \perp, \perp, current\_view]\rangle$ is [U]Rcast (line 34).

The BB-Deliver($\langle m, CI\rangle$) procedure is called when a message is [U]Rdelivered. The protocol proceeds as follows, depending on the value of $CI.type$.

- If $CI.type = ApplMsg$ (line 37), then m is stored in $received_p$ and removed from $pending_p$ (lines 38-39). Furthermore, if the protocol is running within the current sequencer process, which is verified by means of the $Sequencer?(current\_view)$ function (line 40), it increases the $seqnum$ variable (line 41) and assigns it as the sequence number of m, by [U]Rcasting the pair $\langle \perp, [SeqNum, id(m), seqnum, current\_view]\rangle$ (line 42).
- If $CI.type = SeqNum$ message (line 43), then the pair $\langle CI.id, CI.seqnum\rangle$ is stored in $received\_seqs_p$ (line 44).

Finally, the Deliver() procedure is called to start todelivering received messages (line 45).

6.2.2 Send-Broadcast fixed sequencer protocol

The pseudo-code of SB procedures is shown in Figure 14.

When SB-TOcast(m) is called, the pair $\langle m, [\perp, \perp, \perp, current\_view]\rangle$ is sent to the current sequencer, which is determined by applying the function $sequencer()$ to the current view (line 34).

When SB-Deliver($\langle m, CI\rangle$) is called, the protocol (i) removes m from $pending_p$ (line 36), (ii) stores m in $received_p$ (line 37), (iii) stores the pair $\langle CI.id, CI.seqnum\rangle$ in $received\_seqs_p$ (line 38), and (iv) calls the Deliver() procedure to start todelivering received messages (line 39).

Finally, when SB-Receive($\langle m, CI\rangle, q$) is called, the protocol first verifies if it is running within the current sequencer process (line 41). In the affirmative, it increases the $seqnum$ vari-

```
SB FIXED SEQUENCER PROTOCOL()
33  procedure SB-TOCAST(m)
34    Rsend(⟨m, [⊥, ⊥, ⊥, current_view]⟩) to sequencer(current_view);
35  procedure SB-DELIVER(⟨m, CI⟩)
36    pending_p ← pending_p \ {m};
37    received_p ← received_p ∪ {m};
38    received_seqs_p ← received_seqs_p ∪ {⟨CI.id, CI.seqnum⟩};
39    DELIVER();
40  procedure SB-RECEIVE(⟨m, CI⟩, q)
41    if (Sequencer?(current_view)) then
42      seqnum ← seqnum + 1;
43      [U]Rcast(⟨m, [⊥, id(m), seqnum, current_view]⟩);
44  procedure SB-RECOVER()
45    for each m ∈ pending_p
46      Rsend(⟨m, [⊥, ⊥, ⊥, current_view]⟩) to sequencer(current_view);
```

Figure 14: Pseudo-code of the SB fixed sequencer protocol

able (line 42) and assigns it as the sequence number of m by [U]Rcasting the pair $\langle m, [\bot, id(m), seqnum, current\_view]\rangle$ (line 43).

6.2.3 Ask-Broadcast fixed sequencer protocol

The pseudo-code of AB procedures is shown in Figure 15.

When AB-TOCAST(m) is called, the pair $\langle \bot, [GetSeq, id(m), \bot, current\_view]\rangle$ is sent to the current sequencer (line 34).

When AB-RECEIVE($\langle m, CI\rangle, q$) is called, the protocol proceeds as follows, depending on the value of $CI.type$.

- If $CI.type = GetSeq$ (line 42) then the protocol first verifies if it is running within the current sequencer (line 43). In the affirmative, it increases the *seqnum* variable (line 44) and assigns it as the sequence number for the message whose identifier is stored in $CI.id$ by Rsending the pair $\langle \bot, [SeqNum, CI.id, seqnum, current\_view]\rangle$ to q (line 45).
- If $CI.type = SeqNum$ (line 46), then the protocol retrieves from $pending_p$ the message m' with $id(m') = CI.id$ (line 47), and finally [U]Rcasts the pair $\langle m', [\bot, CI.id, CI.seqnum, current\_view]\rangle$ (line 48).

The AB-DELIVER($\langle m, CI\rangle$) procedure works exactly as the SB-DELIVER procedure of the SB fixed sequencer protocol.

6.3 Enforced TO specifications

In this section we exploit the methodology defined in Section 4.2 to identify the TO specification enforced by each fixed sequencer protocol assuming that the system satisfies some synchrony assumptions, thus allowing the VSC layer to enforce liveness.

Let us note that varying the communication pattern (BB, SB and AB) and the used broadcast primitive of the VSC layer (*Rcast*/*Rdeliver* and *URcast*/*URdeliver*), we obtain eight different

```
AB FIXED SEQUENCER PROTOCOL()
33  procedure AB-TOCAST(m)
34    Rsend(⟨⊥, [GetSeq, id(m), ⊥, current_view]⟩) to sequencer(current_view);
35  procedure AB-DELIVER(⟨m, CI⟩)
36    pending_p ← pending_p \ {m};
37    received_p ← received_p ∪ {m};
38    received_seqs_p ← received_seqs_p ∪ {⟨CI.id, CI.seqnum⟩};
39    DELIVER();
40  procedure AB-RECEIVE(⟨m, CI⟩, q)
41    switch
42      case CI.type = GetSeq :
43            if (Sequencer?(current_view)) then
44              seqnum ← seqnum + 1;
45              Rsend(⟨⊥, [SeqNum, CI.id, seqnum, current_view]⟩) to q;
46      case CI.type = SeqNum :
47            MESSAGE m' ← m'' : m'' ∈ pending ∧ id(m'') = CI.id;
48            [U]Rcast(⟨m', [⊥, CI.id, CI.seqnum, current_view]⟩);
49  procedure SB-RECOVER()
50    for each m ∈ pending_p
51      Rsend(⟨⊥, [GetSeq, id(m), ⊥, current_view]⟩) to sequencer(current_view);
```

Figure 15: Pseudo-code of the AB fixed sequencer protocol

fixed sequencer protocols. Following step 1 of the methodology of Section 4.2, we first show that each of these protocols enforces at least $TO(NUA, WNUTO)$. Then, applying steps 2-4 we find their enforced TO specification, according to Definition 1 given in Section 4.2 and following the considerations of Section 5.2. Table 5 anticipates the results. Note that BB protocols are based on two broadcasts, which can be performed using different primitives of the VSC layer. The used communication primitives are reported in the form $first\_broadcast/second\_broadcast$ in Table 5.

Ordering protocol	Communication primitive	TO specification
Broadcast-broadcast sequencer	$Rcast/Rcast$	$TO(NUA, WNUTO)$
	$URcast/URcast$	$TO(UA, SUTO)$
	$Rcast/URcast$	$TO(NUA, WUTO)$
	$URcast/Rcast$	$TO(UA, WNUTO)$
Send-broadcast sequencer	$Rcast$	$TO(NUA, WNUTO)$
	$URcast$	$TO(UA, SUTO)$
Ask-broadcast sequencer	$Rcast$	$TO(NUA, WUTO)$
	$URcast$	$TO(UA, SUTO)$

Table 5: TO specification enforced by each ordering protocol

6.3.1 Preliminary lemmas

Let us recall that messages received from the VSC layer in a view different from the one in which they were sent are filtered out by fixed sequencer protocols (see lines 28 and 31 of Figure 12). As a consequence, the following observation holds.

Observation 1. *The Total Order layer processes only messages that are delivered to it in the same view in which they were sent.*

Since the VSC layer guarantees that messages it delivers are totally ordered with respect to view changes (see Section 5.1), it follows that, to enforce a total order on message deliveries the Total Order layer has to impose an order on messages delivered within each single view. In the following lemmas and theorems we assume that processes belong to the same view v, unless stated otherwise.

Lemma 3. *For any fixed sequencer protocol, if a process p todelivers a message m, then p has executed $[U]Rdeliver(\langle m, CI\rangle)$.*

Proof. Let p be a process todelivering a message m. From the pseudo-code of Figure 12, it follows that this can happen either at statement 9 or at statement 16. In both cases, todelivered messages are taken from $received_p$. As a consequence, at the time of todelivery, m was stored in $received_p$. From the pseudo-codes of Figures 13, 14 and 15, it follows that the X-DELIVER($\langle m, CI\rangle$) procedure must be executed in order to store m in $received_p$. From Figure 12, it follows that p must have executed $[U]Rdeliver(\langle m, CI\rangle)$. □

Lemma 4. *For any fixed sequencer protocol, if a process p executes $[U]Rdeliver(\langle m, CI\rangle)$, then p eventually todelivers m, unless it crashes.*

Proof. We consider the BB case. The proofs for SB and AB cases are similar. Let p be a process executing $[U]Rdeliver(\langle m, CI\rangle)$. First assume that no process crashes. In this case, the VSC layer guarantees that communications are reliable. Therefore, for each tocast message m', every process eventually executes $[U]Rdeliver(\langle m', CI\rangle)$ and $[U]Rdeliver(\langle \perp, [SeqNum, id(m'), seq(m')]\rangle)$. As a consequence, p stores all messages up to m in $received_p$ and their sequence numbers in $received\_seqs_p$ (lines 38 and 44). Therefore, the condition at line 8 is eventually true for m, and it will be eventually todelivered at statement 9.

Now assume that some process crashes. The VSC layer guarantees that eventually p executes statement 13, receiving a new view. Therefore, p eventually todelivers m at statement 16. □

Lemma 5. *For any fixed sequencer protocol using only URcast, if a process p todelivers a message m at statement 9, then any other process todelivering m does so at statement 9.*

Proof. By contradiction. Let p be a process todelivering m at statement 9. From the condition of line 8, it follows that p has also todelivered each message m' such that $seq(m') < seq(m)$. From Lemma 3, it follows that p has executed $[U]Rdeliver(\langle m', CI\rangle)$ for each m'. Furthermore, p must have URdelivered each pair $\langle x, CI\rangle$, with $CI.seqnum = seq(m')$. Let q be a process todelivering m at statement 16. It follows that q has executed statement 13. From properties VSC5 and VSC6, it follows that q must have URdelivered all messages URdelivered by p. As a consequence, all messages up to m are stored in $received_q$ and their sequence numbers are stored in $received\_seqs_q$ before the execution of statement 13. Therefore, the condition of line 8 for m is true at q and it todelivers m at statement 9. Contradiction. □

Corollary 2. *For any fixed sequencer protocol using only Rcast, if a correct process p todelivers a message m at statement 9, then any other correct process todelivers m at statement 9.*

Proof. Trivially follows from Lemma 5, noting that *Rcast* enforces the same guarantees of *URcast* when considering only correct processes. □

Lemma 6. *For any fixed sequencer protocol, if p, q execute statement 13 in view v, then $received_p = received_q$ and $received\_seqs_p = received\_seqs_q$ when this happens.*

Proof. Let p, q be two processes executing statement 13 in view v. From properties VSC5 and VSC6, it follows that they have executed $[U]Rdeliver(\langle m, CI\rangle)$ for the same set of pairs $\langle m, CI\rangle$. As a consequence, for each message m, m is stored in $received_p$ and $received_q$, and $\langle id(m), seq(m)\rangle$ is stored in $received\_seqs_p$ and $received\_seqs_q$. Furthermore, the condition of line 8 is true at p and q for the same set of messages. Therefore p and q todeliver the same set of messages at statement 9 before executing statement 13. As a consequence, $received_p = received_q$ and $received\_seqs_p = received\_seqs_q$ when p and q execute statement 13. □

Corollary 3. *For any fixed sequencer protocol, processes todelivering messages at statement 16 todeliver these messages in the same order.*

Proof. Trivially follows Lemma 6 and from the determinism of function Φ. □

6.3.2 Fixed sequencer protocols enforce at least $TO(NUA, WNUTO)$

Theorem 1. *For each static sub-run, any fixed sequencer protocol enforces NUV.*

Proof. We consider the AB fixed sequencer protocol. The proofs for the other two cases are similar. Let p be a process tocasting a message m. Therefore p stores m in $pending_p$ (line 25), calls AB-TOcast(m) and thus Rsends the pair $\langle \perp, [GetSeq, id(m), \perp, current\_view]\rangle$ to the sequencer (line 34). In the absence of failures, from property VSC1, it follows that the sequencer Rreceives this pair. It will then Rsend the pair $\langle \perp, [SeqNum, id(m), seq(m), current\_view]\rangle$ to p (line 45). Property VSC1 guarantees that p Rreceives this pair. It will then [U]Rcast the pair $\langle m, [\perp, id(m), seq(m), current\_view]\rangle$ (line 48). Property VSC3 guarantees that p [U]Rdelivers this pair. From Lemma 4, it follows that p todelivers m.

In case of failures, the VSC layer guarantees that a view change eventually occurs. As a consequence, p Rsends m to the new sequencer upon the delivery of the view change event (line 51). Since p is correct and since we are assuming a finite set of processes that do not recover from failures, p will eventually install a view containing only correct processes (only p in case all other processes are faulty), and it will therefore succeed in todelivering m. □

Theorem 2. *For each static sub-run, any fixed sequencer protocol enforces UI.*

Proof. We consider the BB fixed sequencer protocol. The proofs for the other two cases are similar. Let p be a process todelivering a message m. From Lemma 3, it follows that p has executed $[U]Rdeliver(\langle m, CI\rangle)$. From property VSC4, it follows that there exists a process q that has executed $[U]Rcast(\langle m, CI\rangle)$ (line 34). As a consequence, a process q has tocast m.

Now assume by contradiction that p todelivers m twice and let q be the sender of m. From Lemma 3, it follows that p has executed $[U]Rdeliver(\langle m, CI\rangle)$. Consider the first time m is todelivered by p (either at statement 9 or at statement 16). In both cases, m is deleted from $received_p$. Therefore, p must have executed $[U]Rdeliver(\langle m, CI\rangle)$ twice. From property VSC4, it follows that q has executed $[U]Rcast(\langle m, CI\rangle)$ twice. From the pseudo-code of Figure 13, it follows that q can execute $[U]Rcast(\langle m, CI\rangle)$ either at statement 34 or at statement 48. If q executes $[U]Rcast(\langle m, CI\rangle)$ at statement 48, then m was stored in $pending_q$. Since messages are stored in $pending_q$ upon the execution of TOcast() at q, which in turn makes q execute statement 34, it follows that q executes $[U]Rcast(\langle m, CI\rangle)$ for the first time at statement 34.

As each message is tocast at most once, it follows that q executes $[U]Rcast(\langle m, CI\rangle)$ for the second time at statement 48. Let v be the view in which p executes $[U]Rdeliver(\langle m, CI\rangle)$ for the first time. From line 28, it follows that q has executed its first $[U]Rcast(\langle m, CI\rangle)$ in view v. Furthermore, in order to execute line 48, q must have executed line 13 in view v. From properties VSC5 and VSC6, it follows that q has executed $[U]Rdeliver(\langle m, CI\rangle)$ and thus it has also executed line 39. Therefore m is deleted from $pending_q$ before the execution of line 13 and q cannot execute $[U]Rcast(\langle m, CI\rangle)$ at line 48. Contradiction. □

Theorem 3. *For each static sub-run, any fixed sequencer protocol enforces NUA.*

Proof. Let p be a correct process todelivering message m. From Lemma 3, it follows that p has executed $[U]Rdeliver(\langle m, CI\rangle)$. From property VSC5, it follows that any correct process q will execute $[U]Rdeliver(\langle m, CI\rangle)$. From Lemma 4, it follows that q eventually todelivers m. □

Theorem 4. *For each static sub-run, any fixed sequencer protocol enforces $WNUTO$.*

Proof. By contradiction. Let p, q be two correct processes both todelivering messages m, m'. Assume by contradiction that $WNUTO$ is not satisfied. Therefore, without lack of generality, assume that p todelivers m before $m'$and q todelivers m' before m. From Corollary 2, it follows that there are two possible cases:

m **and** m' **are todelivered at statement 9.** From the condition of line 8 at p, it follows that $seq(m) > seq(m')$, whereas from the condition of line 8 at q, it follows that $seq(m) < seq(m')$. Contradiction.

m **and** m' **are todelivered at statement 16.** This contradicts Corollary 3.

□

Theorem 5. *Any fixed sequencer protocol enforces at least $TO(NUA, WNUTO)$.*

Proof. It follows from Theorems 1, 2, 3 and 4. □

6.3.3 Fixed sequencer protocols using only $URcast$ enforce $TO(UA, SUTO)$

Theorem 6. *For each static sub-run, any fixed sequencer protocol using only $URcast$ enforces UA.*

Proof. Let p be a process todelivering message m. From Lemma 3, it follows that p has executed $URdeliver(\langle m, CI\rangle)$. From the guarantees of the VSC layer, it follows that any correct process q will execute $URdeliver(\langle m, CI\rangle)$. From Lemma 4, it follows that q eventually todelivers m. □

Theorem 7. *For each static sub-run, any fixed sequencer protocol using only $URcast$ enforces $SUTO$.*

Proof. By contradiction. Let p be a process todelivering m, m' in this order, and let q be a process todelivering m'. Assume by contradiction that $\neg SUTO$ is satisfied. Then either q does not todeliver m, or it todelivers m after m' (see also Table 3). We can have the following cases:

q **todelivers** m' **at statement 9.** From the condition of line 8, it follows that q has todelivered all messages with sequence number less than $seq(m')$. Therefore $seq(m) > seq(m')$ (provided that m succeeds in obtaining a sequence number from the sequencer). From Lemma 5, it follows that p has todelivered m' at statement 9. From the pseudo-code of Figure 12, it follows that p has also todelivered m at statement 9 and that $seq(m) < seq(m')$. Contradiction.

q **todelivers** m' **at statement 16.** In this case, q must have executed statement 13. From Lemma 3, it follows that p has executed $URdeliver(\langle m, CI\rangle)$. From the properties VSC5 and VSC6, it follows that q must have executed $URdeliver(\langle m, CI\rangle)$ before executing statement 13. Therefore $m \in received_q$ when q executes statement 13. As a consequence, q todelivers m after m' at statement 16. From Lemma 5, it follows that also p todelivers m, m' at statement 16. This contradicts Corollary 3.

□

Theorem 8. *Any fixed sequencer protocol using only URcast enforces* $TO(UA, SUTO)$.

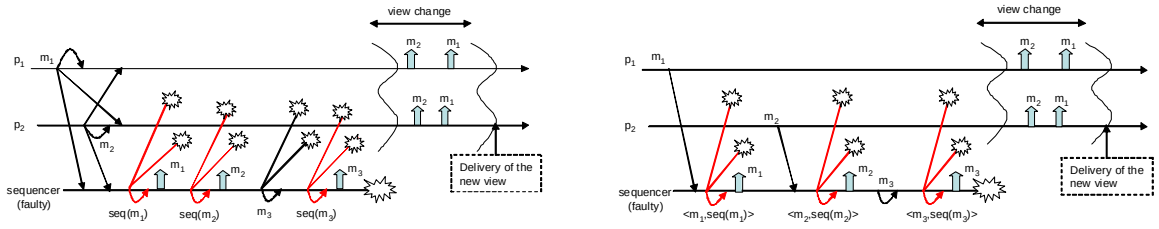
Proof. It follows from Theorem 1, 2, 6 and 7. □

We now apply the methodology of Section 4.2 to identify the TO specification enforced by the remaining variations of fixed sequencer protocols.

6.3.4 Specifications enforced by Broadcast-Broadcast fixed sequencer protocols

We denote BB fixed sequencer protocols $BB(first\_broadcast, second\_broadcast)$ to distinguish the communication primitives of the VSC layer that are used to perform the first and second broadcast of the BB communication pattern.

Theorem 8 shows that $BB(URcast, URcast)$ enforces $TO(UA, SUTO)$. In the following we analyze remaining BB fixed sequencer protocols.



(a) A run satisfying $\neg WUTO \wedge \neg SUTO \wedge \neg UA$ generated by a BB fixed sequencer protocol using only *Rcast*

(b) A run satisfying $\neg WUTO \wedge \neg SUTO \wedge \neg UA$ generated by a SB fixed sequencer protocol using only *Rcast*

Figure 16: Runs generated by BB and SB fixed sequencer protocols using only *Rcast*

Theorem 9. $BB(Rcast, Rcast)$ *enforces* $TO(NUA, WNUTO)$.

Proof. From Theorem 5, it follows that we can go to step 2 of the methodology of Section 4.2. Figure 16(a) shows a run generated by $BB(Rcast, Rcast)$. This run satisfies $\neg UA \wedge \neg SUTO \wedge \neg WUTO$. As a consequence, by applying step three of the methodology, it follows that $BB(Rcast, Rcast)$ enforces $TO(NUA, WNUTO)$. □

Theorem 10. *For any static sub-run,* $BB(Rcast, URcast)$ *enforces* $WUTO$.

Proof. By contradiction. Let p, q be two processes and let m, m' be two messages both todelivered by p and q. Assume by contradiction that $\neg WUTO$ is satisfied. Therefore, without lack of generality, assume that p todelivers m before m' and q todelivers m' before m (see Table 3). From Lemma 3, it follows that both p and q have executed $Rdeliver(\langle m, CI\rangle)$ and $Rdeliver(\langle m', CI'\rangle)$. Therefore m and m' are stored in $received_p$ and $received_q$. Furthermore, from Corollary 3, it follows that at least one process has todelivered at least one message at statement 9. Without lack of generality, let p be this process and m be this message. From the condition of line 8 at p, it follows that $seq(m)$ is stored in $received\_seqs_p$. Therefore p has URdelivered the tuple $\langle \perp, [SeqNum, id(m), seq(m), current\_view]\rangle$. Furthermore, $seq(m') > seq(m)$ (provided that m' succeeds in obtaining a sequence number). Then we can have two cases:

q todelivers m at statement 9. From the condition of line 8 at q, it follows that $seq(m') < seq(m)$. Contradiction.

q todelivers m at statement 16. In that case, q must have executed statement 13. From properties VSC5 and VSC6, it follows that $\langle \perp, [SeqNum, id(m), seq(m), current\_view]\rangle$ is stored in $received\_seqs_q$ before the execution of statement 13. From the definition of the Φ function, it follows that $seq(m') < seq(m)$. Contradiction.

□

Theorem 11. $BB(Rcast, URcast)$ *enforces* $TO(NUA, WUTO)$.

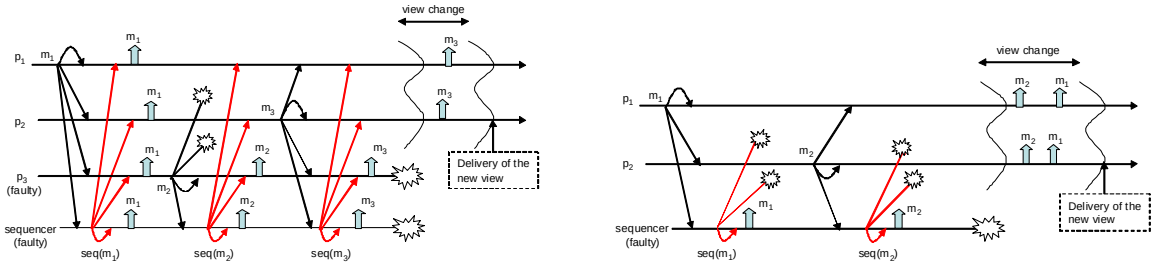
Proof. From Theorem 5, it follows that we can go to step 2 of the methodology of Section 4.2. Figure 17(a) shows a run generated by $BB(Rcast, URcast)$. This run satisfies $WUTO \wedge \neg UA \wedge \neg SUTO$. As a consequence, we apply step three of the methodology twice. The next step is to check if $\neg WUTO$ can be satisfied. From Theorem 10, it follows that this is not possible. As a consequence, $BB(Rcast, URcast)$ enforces $TO(NUA, WUTO)$. □

Theorem 12. *For any static sub-run,* $BB(URcast, Rcast)$ *enforces* UA.

Proof. Same as Theorem 6. □

Theorem 13. $BB(URcast, Rcast)$ *enforces* $TO(UA, WNUTO)$.

Proof. From Theorem 5, it follows that we can go to step 2 of the methodology of Section 4.2. Figure 17(b) shows a run generated by $BB(Rcast, URcast)$. This run satisfies $UA \wedge \neg SUTO \wedge \neg WUTO$. As a consequence, we apply step three of the methodology twice. The next step is to check if $\neg UA$ can be satisfied. From Theorem 12, it follows that this is not possible. As a consequence, $BB(URcast, Rcast)$ enforces $TO(UA, WNUTO)$. □



(a) A run satisfying $WUTO \wedge \neg SUTO \wedge \neg UA$ generated by $BB(Rcast, URcast)$

(b) A run satisfying $\neg WUTO \wedge \neg SUTO \wedge UA$ generated by $BB(URcast, Rcast)$

Figure 17: Runs generated by BB fixed sequencer protocols using both $Rcast$ and $URcast$

6.3.5 Specifications enforced by Sequencer-Broadcast fixed sequencer protocols

From Theorem 8 it follows that the SB protocol using $URcast$ enforces $TO(UA, SUTO)$. The following Theorem shows that SB protocol using $Rcast$ enforces $TO(NUA, WNUTO)$.

Theorem 14. *SB sequencer protocol using Rcast enforces* $TO(NUA, WNUTO)$

Proof. From Theorem 5, it follows that we can go to step 2 of the methodology of Section 4.2. Figure 16(b) shows a run generated SB sequencer protocol using $Rcast$. This run satisfies $\neg UA \wedge \neg SUTO \wedge \neg WUTO$. As a consequence, by applying step three of the methodology, it follows that SB sequencer protocol using $Rcast$ enforces $TO(NUA, WNUTO)$. □

6.3.6 Specifications enforced by Ask-Broadcast fixed sequencer protocols

From Theorem 8 it follows that the AB protocol using $URcast$ enforces $TO(UA, SUTO)$. In the following we analyze the AB protocol using $Rcast$.

Theorem 15. *For each static sub-run, AB protocol using Rcast enforces* $WUTO$.

Proof. By contradiction. Let p, q be two processes and let m, m' be two messages both todelivered by p and q. Assume by contradiction that $\neg WUTO$ is satisfied. Therefore, without lack of generality, assume that p todelivers m before m' and q todelivers m' before m (see Table 3). From Lemma 3, it follows that both p and q have executed $Rdeliver(\langle m, CI \rangle)$ and $Rdeliver(\langle m', CI' \rangle)$. Furthermore, from the pseudo-code of Figure 15, it follows that $CI.seqnum = seq(m)$ and $CI'.seqnum = seq(m')$. Therefore m and m' are stored in $received_p$ and $received_q$, and $\langle id(m), seq(m) \rangle$ and $\langle id(m'), seq(m') \rangle$ are stored in $received\_seqs_p$ and $received\_seqs_q$. From the condition of line 8 at p and the definition of the Φ function, it follows that $seq(m) < seq(m')$. From the condition of line 8 at q and the definition of the Φ function, it follows that $seq(m') < seq(m)$. Contradiction. □

Theorem 16. *AB sequencer protocol using Rcast enforces* $TO(NUA, WUTO)$.

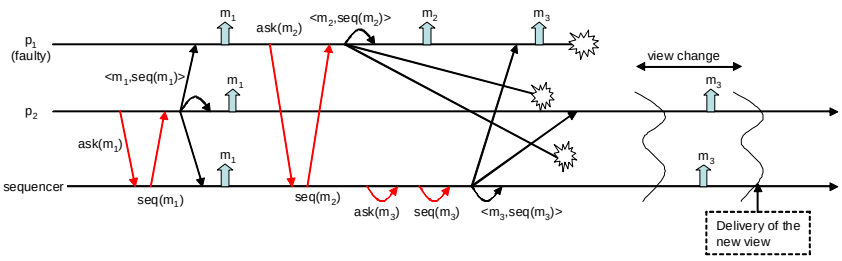


Figure 18: A run satisfying $WUTO \wedge \neg SUTO \wedge \neg UA$ generated by an AB fixed sequencer protocol

Proof. From Theorem 5, it follows that we can go to step 2 of the methodology of Section 4.2. Figure 18 shows a run generated by the AB fixed sequencer protocol using *Rcast*. This run satisfies $WUTO \wedge \neg UA \wedge \neg SUTO$. As a consequence, we apply step three of the methodology twice. The next step is to check if $\neg WUTO$ can be satisfied. From Theorem 15, it follows that this is not possible. As a consequence, AB sequencer protocol using *Rcast* enforces $TO(NUA, WUTO)$. □

7 Related work

Several other works address the problem of organizing the great amount of literature about TO communications in a coherent framework, e.g. [14], [16] and [27] just to name some of the most relevant. These papers do not address the impact of different TO specifications on application logics by a detailed analysis of the possible behaviors of processes under failure scenarios. Furthermore, Chockler *et al.* [14] and Wilhelm and Schiper [27] present a set of TO specifications explicitly bound to view-oriented group communication systems. In contrast, we adopted more abstract specifications (following the approach of Hadzilacos and Toueg [19]) that are not bound to any specific class of systems.

In particular, Wilhelm and Schiper [27] identify four distinct TO specifications and organizes them into a hierarchy. Following the considerations presented in Section 5.2, this hierarchy is a sub-hierarchy of the one presented in this paper having $TO(UA, WUTO)$ as the root node. Let us note that $TO(UA, WUTO)$ and $TO(NUA, WNUTO)$ have been also introduced in several other papers, e.g. [19]. Furthermore, our hierarchy incorporates $TO(UA, SUTO)$, which was first introduced by Aguilera *et al.* [1], as well as new specifications, i.e. $TO(NUA, SUTO)$, $TO(NUA, SNUTO)$ and $TO(UA, SNUTO)$. The latter two specifications have been formally proved to be equivalent to $TO(NUA, WNUTO)$ and $TO(UA, WNUTO)$, respectively.

Differently from this paper and from [27], Chockler *et al.* [14] pursue the objective of systematically formalizing all the properties exhibited by a wide class of view-oriented group communication systems. As a consequence, the TO specifications presented in this work are tightly coupled to the properties enforced by the layers commonly underlying TO implementations in this class of systems. In this context, three distinct Total Order properties, namely Weak Total Order, Strong Total Order and Reliable Total Order, which also apply to partitionable systems are discussed. These properties correspond in some way to $WNUTO$, $WUTO$ and $SUTO$ respectively. Let us remark that [14] also considers the semantics of total order communications to

multiple groups and with differentiated message types, which are out of the scope of this paper.

Another relevant work is [16] (a more recent version of the original survey presented in [15]), which describes more than fifty TO implementations, including those that assume malicious fault models (which are not considered in this paper). Implementations are classified according to TO specifications very similar to the ones presented in this paper, but the behavior of faulty processes admitted by each specification, as well as their impact on applications, is not discussed.

It is also worth noting that the use of first order logic to formalize the specification of group communication systems is not new. As examples, Babaoğlu *et al.* [2], as well as Wilhelm and Schiper [27] exploit, to some extent, notations similar to those used in this paper. Let us note that this work is complemented by a companion paper [5] which exploits the classification framework and the methodology presented in Section 4.2 to identify the TO specification implemented by several group communication systems, also comparing them from a performance point of view.

Finally, this paper has not addressed the problem of constraint processes to deliver messages according to the order of their sendings. Therefore *fifo* and *causal atomic broadcast* specifications [22] are out of the aim of this paper. However, such specifications can be easily defined by adding a specific ordering property [19] to the ones defined in Section 3 and implementations can be realized by adding simple mechanisms based on scalar and vector clocks [4] to the protocols presented in this paper.

8 Conclusion

In this paper we presented in a systematic way eight TO specifications using a well-defined formal framework based on first order logic. We then proved that some of them are equivalent, thus giving rise to six significant TO specifications. These specifications are organized into a hierarchy based on a precedence relation that actually models the inclusion between sets of runs admitted by distinct specifications. Differences among these sets of runs, as well as between the behaviors of faulty and correct processes admitted by each specification have been characterized. Furthermore, a simple methodology that enables the classification of TO implementations in the hierarchy has been proposed and applied in the context of view-oriented group communication systems. In particular, eight variations of fixed sequencer protocols have been formally analyzed to identify their enforced specification. Interestingly, this eight variations are able to enforce only four out of the six TO specifications in the hierarchy.

This paper is mainly intended for practitioners who want to exploit TO primitives to build distributed applications. As such, our discussion has the primary goal of explaining the subtleties hidden in the behavior of a TO implementation under failure scenarios, and their impact on the correctness of an application built upon it. Therefore, other than providing a formal classification of TO specifications, we also explicitly discuss the possible scenarios that each specification allows to occur and their implications on the design of application logics.

Acknowledgements

We want to thank the anonymous reviewers for their effective comments and suggestions, which helped us to improve the the presentation and the overall quality of this work.

References

[1] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Thrifty generic broadcast. In *Proceedings of the 14th International Symposium on Distributed Computing (DISC)*, number 1914, pages 268–282, Toledo, Spain, 2000. Springer-Verlag.

[2] Ö. Babaoğlu, R. Davoli, and A. Montresor. Group membership and view synchrony in partitionable asynchronous distributed systems: Specifications. Technical Report UBLCS–95–18, Department of Computer Science, University of Bologna, Italy, 1995.

[3] R. Baldoni, S. Cimmino, and C. Marchetti. Total order communications over asynchronous distributed systems: specifications and implementations. Technical Report 06/04, Università di Roma "La sapienza", January 2004.

[4] R. Baldoni and M. Raynal. Fundamentals of Distributed Computing: A Practical Tour of Vector Clock Systems. *IEEE Distributed Systems Online*, 3(2), 2002.

[5] Roberto Baldoni, Stefano Cimmino, and Carlo Marchetti. Total order communications: a practical analysis. In *Proceedings of the Fifth European Dependable Computing Conference (EDCC-5)*, pages 38–54, Budapest, Hungary, April 2005.

[6] B. Ban. Design and implementation of a reliable group communication toolkit for java. Cornell University, September 1998.

[7] A. Basu, B. Charron-Bost, and S. Toueg. Simulating Reliable Links with Unreliable Links in the Presence of Process Crashes. In *Proceedings of Distributed Algorithm (WDAG '96)*, pages 105–122, October 1996.

[8] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proceedings of the 11th ACM Symp. on Operating Systems Principles*, pages 123–138, December 1987.

[9] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[10] K. Birman and R. van Renesse. *Reliable Distributed Computing with the ISIS toolkit*. IEEE CS Press, 1994.

[11] K. P. Birman. A Review of Experiences with Reliable Multicast. *Software – Practice and Experience*, 29(9):741–774, 1999.

[12] T. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the Impossibility of Group Membership. In *Proc. of the 15th ACM Symposium of Principles of Distributed Computing*, 1996.

[13] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving Consensus. *Journal of ACM*, 43(4):685–722, July 1996.

[14] G.V. Chockler, I. Keidar, and R. Vitenberg. Group Communications Specifications: a Comprehensive Study. *ACM Computing Surveys*, 33(4):427–469, December 2001.

[15] X. Dèfago. *Agreement-Related Problems: From Semi Passive Replication to Totally Ordered Broadcast*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 2000. PhD thesis no. 2229.

[16] X. Dèfago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. Technical Report IC/2003/56, École Polytechnique Fédérale de Lausanne, Switzerland, September 2003.

[17] H. Garcia-Molina and A. Spauster. Ordered and reliable multicast communication. *ACM Transactions on Computer Systems*, 9(3):242–271, August 1991.

[18] R. Guerraoui and A. Schiper. Software-Based Replication for Fault Tolerance. *IEEE Computer - Special Issue on Fault Tolerance*, 30:68–74, April 1997.

[19] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcast and Related Problems. In S. Mullender, editor, *Distributed Systems*, chapter 16. Addison Wesley, 1993.

[20] M.G. Hayden. *The Ensemble System*. PhD thesis, Cornell University, Ithaca, NY, USA, 1998.

[21] Jason Hickey, Nancy Lynch, and Robbert van Renesse. Specifications and proofs for ensemble layers. In R. Cleaveland, editor, *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 1579, pages 119–133. Springer-Verlag, Berlin Germany, 1999.

[22] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[23] Sergio Mena, André Schiper, and Paweł Wojciechowski. A step towards a new generation of group communication systems. Technical Report IC/2003/01, École Polytechnique Fédérale de Lausanne, Switzerland, January 2003.

[24] D. Powell. Group Communication. *Communications of the ACM*, 39(4):50–97, 1996.

[25] A. Schiper. Dynamic group communication. Technical Report IC/2003/27, École Polytechnique Fédérale de Lausanne, Switzerland, April 2003.

[26] Fred B. Schneider. Replication Management Using the State Machine Approach. In S. Mullender, editor, *Distributed Systems*. ACM Press - Addison Wesley, 1993.

[27] U. G. Wilhelm and A. Schiper. A hierarchy of totally ordered multicasts. In *Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems (SRDS-14)*, Bad Neuenahr, Germany, September 1995.

Appendix A: Formal analysis of the behavior of faulty processes

For each specification we derive a predicate characterizing all admitted differences between faulty and correct processes with respect to their sets of delivered messages. First we define (by inspection) the predicate, denoted $\Delta(UA, SUTO)$, characterizing the differences admitted

TO specification	$\Delta(A, O)$
$TO(UA, SUTO)$	$(\exists m\ todel(q, m) \wedge \neg todel(p, m) \wedge \nexists m'\ todel(p, m) < todel(p, m'))$
$TO(UA, WUTO)$	$\Delta(UA, SUTO) \vee (\exists m, m'\ todel(q, m) < todel(q, m') \wedge todel(p, m') \wedge \neg todel(p, m))$
$TO(UA, WNUTO)$	$\Delta(UA, WUTO) \vee (\exists m, m'\ todel(p, m) < todel(p, m') \wedge todel(q, m') < todel(q, m))$
$TO(NUA, SUTO)$	$\Delta(UA, SUTO) \vee (\exists m\ todel(p, m) \wedge \neg todel(q, m))$
$TO(NUA, WUTO)$	$\Delta(UA, WUTO) \vee \Delta(NUA, SUTO)$
$TO(NUA, WNUTO)$	$\Delta(NUA, WUTO) \vee \Delta(UA, WNUTO)$

Table 6: Admitted differences among the set of messages delivered by faulty and correct processes (q is correct and p is faulty)

by $TO(UA, SUTO)$. Then, thanks to the transitivity of relation $\rightarrow$ and to the specification hierarchy, we derive all other predicates by using the following recurrence formula:

$$\begin{array}{rl} \Delta(A,O) = & (\bigvee_{\forall\ TO(A',O')\quad TO(A,O)} \Delta(A',O')) \vee \\ & (\bigvee_{\forall\ TO(A',O)\quad TO(A,O)} TO(A,O) \wedge \neg A') \vee \\ & (\bigvee_{\forall\ TO(A,O')\quad TO(A,O)} TO(A,O) \wedge \neg O') \end{array}$$

where $\rightsquigarrow$ is the the transitive reduction of relation $\rightarrow$, depicted in Figure 5.
This formula derives from the following observations.

1. Due to the transitivity of the $\rightarrow$ relation, a specification $TO(A, O)$ allows all the execution patterns (and thus the differences among the sets of messages delivered by faulty and correct processes) admitted by each specification $TO(A', O') : TO(A', O') \rightarrow TO(A, O)$. This is captured by the first term of the formula, i.e. $\bigvee_{\forall\ TO(A',O')\quad TO(A,O)} \Delta(A', O')$;

2. If $TO(A, O) \rightsquigarrow TO(A', O)$ (respectively $TO(A, O) \rightsquigarrow TO(A, O')$), then the predicate $TO(A', O) \wedge \neg A$ (respectively $TO(A, O') \wedge \neg O$) captures all runs (and thus the differences among the sets of messages delivered by faulty and correct processes) that are admitted by $TO(A', O)$ (respectively $TO(A, O')$) and not by $TO(A, O)$. This accounts for the second and third terms of the formula, i.e. $(\bigvee_{\forall\ TO(A',O)\quad TO(A,O)} TO(A, O) \wedge \neg A') \vee (\bigvee_{\forall\ TO(A,O')\quad TO(A,O)} TO(A, O) \wedge \neg O')$.

By applying the recurrence formula for each specification we obtain a predicate capturing the differences among the sets of messages delivered by faulty and correct processes. Table 6 reports these predicates.

Let us note that, for each specification $TO(A, O)$, the predicate $\Delta(A, O)$ formally describes the execution patterns admitted by $TO(A, O)$, which are reported in Table 4. As an example, $\Delta(UA, SUTO)$ captures the execution pattern EP1 described in Section 4.1.