A networking model for Casanova 2.0

Dr. Giuseppe Maggiore

December 25, 2013

1 Introduction

Adding multi-player support to games is a highly desirable feature. By letting players interact with each other, new forms of gameplay, cooperation, and competition emerge without requiring any additional effort by the developer. This allows a game to remain fresh and playable, even after the single player content has been exhausted.

Multi-player support in games is a very expensive piece of software to build. The requirements of performance mean that many high-level protocols or mechanisms are insufficient, either because they are too slow computationally (for example they may rely on reflection to work properly) or because they transmit too much data across the network (for example the do not support incremental updates). This leads developers to building low level protocols that use UDP (unreliable) transfer, manually synchronizing the game instances one byte at a time. This process by itself is highly expensive. Such a low level protocol is difficult to get right, and debugging subtle protocol mismatches, transmission errors, etc. will take lots of development resources. To make matters worse, players often behave erratically. It is widespread practice among players to leave a competitive game as soon as their defeat is apparent (a phenomenon so common to even have its own name: "rage quitting"), thereby requiring developers to build their system so that it is very resilient to clients leaving the game or even reconnecting afterwards. Also, the requirements of every game vary significantly: from turn-based games that only need to synchronize the game world every few seconds, and where latency is not a big issue, to first-person-shooter games where prediction mechanisms are needed to ensure the smooth movement of synchronized entities, to real-time-strategy games where thousands of units on the screen all need to be synchronized across game instances. In short, very little reuse of previous code or libraries is possible, thereby making previous effort substantially not accessible for new titles.

We observe that networking models and algorithms do not vary substantially between games, even though the code that needs to be written to implement them does. The similarity comes from the fact that the ways to serialize, synchronize, and predict the behaviour of entities are relatively standard and described according to a limited series of general ideas. The difference, on the other hand, comes from the fact that low-level protocols need to be adapted to the specific structure of the game world and the data structures that make it up. Networking code can thus be seen as a series of (parametric) transformations from the data definitions of the game world and its structures, into the code that serializes these structures within the constraints imposed by games. This leads us to believe that the difficulties in networking do not stem from an intrinsic, theoretical or fundamental difficulty, but rather they come from the lack of ability of modern programming languages to capture such complex (inductive) transformations from data structures into algorithms. By designing the proper abstractions for networking at the language level, we aim at simplifying the task of building low-level networking protocols for games. We wish to, at the same time, give the developer more powerful and flexible tools to build networking code, always at the proper level of abstraction. We wish to avoid something that is really low-level, for example just a few byte-sending primitives, because this would not really solve the problem but would instead just offer some ready-made functions to call. At the same time we want to avoid extremely high-level primitives such as `synchronize-world`, because these would take away the flexibility to design ad-hoc protocols for each scenario.

Contribution of this work. In this paper we present a novel approach to networking in games. This approach is built inside the *Casanova 2.0* language, which is designed for real-time applications such as simulations, and games in particular.

2 Problem description

Networking in games presents a series of standard challenges that we now list and discuss. We will use these challenges as the main reference when doing the evaluation of our networking model.

2.1 Set up of connections

A networked game needs to initialize a series of communication channels between (a subset of) the various active instances. Roles between the instances must be specified somehow. For example, whether an instance is a host or a client. Alternatively, it may be needed to specify which game entities are owned by which instances in a peer-to-peer architecture.

2.1.1 Tunnelling

In the very common case when some of the game instances are behind routers, firewalls, or in general not directly accessible, it may also be needed to set up some tunnelling mechanisms that make it possible to route communications through a commonly accessible mediator. This problem is usually solved through the use of a separate *tracking server*, which is publicly accessible on the Internet. This server acts as a mediator, and ensures that all instances of the game can effectively communicate with each other even when they themselves are not directly accessible on the Internet through a static IP or a similar mechanism. Notice that *in this paper we do not worry about this scenario*. We will assume that the local peers can be discovered as if on a LAN, whether directly or thanks to the mediation of some tracking server. This means that we will not focus on the very first steps of a networked game, but rather on the mechanisms of the game itself.

2.2 Incremental updates

Information across instances does not usually go in the form of whole entities synchronized across the network. This would be convenient from the programmer's perspective, but it would not be feasible when bandwidth and latency are limited, which is virtually every realistic scenario. For this reason, all multi-player games usually transmit only user input, incremental updates to entities, and little more. Transmitting whole entities is rarely done, if ever.

2.3 Robustness

The sudden disappearance from the network of an instance of the game is a common phenomenon. This happens because of unexpected, unintentional difficulties such as electrical interruptions, network cable disconnections, etc., but also because of intentional interruptions: a player may lose interest in the game, or need to leave the game to do something else. Even after a player disconnects, the game may need to keep running: either a new player may be added to fill in for the missing one, or the game will go on just for the remaining players. This means that games need to be resilient to loss of a single instance.

2.4 CPU usage

A game needs to be able to present a new, updated picture on the screen at a rate that is fast enough to give a perception of smoothness. This means that computations for the update and drawing of said picture need to be completed before too much time has elapsed. If networking code is excessively expensive computationally, then it will be impossible for the game to run in real-time.

2.5 Bandwidth and latency limitations

Available bandwidth is quite limited. Noisy wireless networks, poor Internet connections, multiple users sharing a domestic connection, etc. mean that the total bandwidth available to an instance of the game is going to be quite limited. Similarly, latency may be significant for some transmissions, especially if a lot of data is being sent.

The game needs to transmit as little data as possible, for example avoiding the transmission of every updated field, rather relying on mechanisms that locally predict the behaviour of remote entities such as interpolation and extrapolation.

3 Approach overview

In this section we describe the general approach of networking in Casanova 2.0. We discuss the architecture, the syntactic and semantic elements, and give a rough idea of how they work. We conclude the section with an in-depth example that shows how to build a small game with a lobby with Casanova 2.0.

3.1 Architecture

The architecture of Casanova networking is peer-to-peer. This choice is motivated by the fact that by nature none of the players is significantly more important than the others, and clients may disconnect (because of faults or purposefully because of their users) at all times. A peer-to-peer architecture also has the advantage that each instance is only responsible for a subset of entities: usually those that were generated locally in that instance. A peer-to-peer architecture is compared to a more traditional host-clients architecture where one of the player is the *game host*. The game host usually contains the latest version of the game world, and acts as the latest, authoritative version of the game which all clients are eventually forced to show on their local screens. The advantages of peer-to-peer are various. Peer-to-peer allows us to naturally build systems where computational load is distributed across the various clients. For example, AI that drives the game entities may be run only for the local entities, while the remote ones spawned on the other clients are just synchronized across the net. Also, if a client has transmission problems, then he will not necessarily disrupt the game for all players. Only interactions with him will be problematic, but interactions with other, accessible clients are still possible. In a host-clients architecture, on the other hand, the host is responsible for most computations (such as AI), so the host machine needs to be significantly more powerful than that of the other players. Also, if the host leaves the game, then the game will suffer from issues that range from just quitting the game for all other players, to lengthy waits as a client is promoted to become the new host, etc.

3.2 Ownership of entities

In Casanova, entities have a concept of ownership. The instance of the game where an entity is constructed is called the *master* of the entity. The instances of the game that received that entity across the net, on the other hand, are called the *slaves* of the entity. This means that each entity can have two sets of rules: one for its master instance, and one for its slave instances. The master rules will usually (but not necessarily) update the internal logic of the entities that are locally owned, and send (some of) the updates to the slaves. The slave rules will usually (but not necessarily) receives the updates from the master and potentially interpolate the values in order to give a perception of smoothness.

3.3 Connection primitives

Casanova supports the concept of connection and disconnection. Each entity may specify what transmissions (or even logic) happens when a new instance of the game connects to the current instances. *Master connections* will usually send the locally owned entity to the new instance. *Slave connections* will usually receive the remotely owned entity from existing instances, and add it to the game world.

3.4 Transmission primitives

In order to be able to send and receive data in a convenient manner, Casanova offers three primitives: `send`, `receive`, `send_reliable`, and `receive_many`. The different versions of *send* are used, respectively, for sending data without confirmation or with confirmation. Confirmation is more expensive in terms of used bandwidth, and can be used when a value *needs* to be transmitted correctly in order to ensure the functioning of the game. The different versions of *receive* are used, respectively, to receive a value from a single instance or from all the current other instances of the game. `receive_all` is akin to primitives such as `gather` in the MPI framework.

Send and receive are generic primitives, meaning that they are capable of full serialization of more complex values. This allows a significant simplification with respect to network libraries which only can transmit elementary data types, because it can be used to hide large and complex transmissions of compound data types without any programmer intervention. This can greatly reduce the amount of bugs that derive from misaligned transmissions such as sending a value on one instance but not receiving it on the other instance.

3.5 Samples

In this section we discuss some samples in order to see the Casanova networking primitives in action. We will only show the internal logic of the samples, and the synchronization primitives and the communication protocols. We purposefully avoid showing any rendering code or complex game logic, as that is out of the scope of this work.

3.5.1 A simple chat

The first example we see is very simple: an unregulated chat where any new instance of the game can directly add a line to the shared chat lines.

The game world, which represents the main data structure that contains the game data and the rules for a whole instance of the game, is made up of only three fields: the name of the local player, the text of the chat so far, and the text that makes up the line that the local player is writing:

```
world Chat = {
  Name                          : string
  Text                          : string
        Line                            : string
```

We now specify a series of rules. Rules define how one (or more) fields are updated as time progress in the game or certain events take place (for example key presses). The main rule that governs the dynamics of the chat game will update the `Line` and `Text` fields; we see this from the rule header:

```
        rule Line,Text =
```

The rule waits until enter is pressed. When this happens, we send the current `Line` prefixed by the local user `Name`. We then reset the local value of `Line` to the empty string, and add that string to the `Text`:

```
    wait_until (IsKeyPressed(Enter))
    send_reliable<string>(Name + ":" + Line)
    yield "", Text + "\n" + Line
```

If the user pressed backspace, and if the `Line` was an empty string, then we remove the last character from the string:

```
        rule Line =
                wait_until (IsKeyPressed(Backspace) && Line
                   <> "")
                yield Line <- Line.Tail
```

Finally, for all character pressures we simply append the typed character to `Line`:

```
        rule Line =
                let k = wait_until (IsCharPressed())
                yield Line <- Line + k
```

There is also a rule that, in parallel to the previous one, waits until a string is received from one of the other clients and appends it to the `Text`:

```
        rule Text =
          yield Text + "\n" + receive<string>()
```

Finally, we specify the `Create` function that initializes the game world. In this case we take as input the local user name, and use it to initialize the `Name` field. All other fields start as empty strings:

```
        Create(own_name) =
                {
                        Name = own_name
                        Text = ""
                        Line = ""
                }
}
```

3.5.2 A game lobby

We now discuss a more complex example: a game lobby (or just *lobby).* A lobby allows a group of players to coordinate, chat, and in general choose game options right before the game starts. The lobby is an especially interesting case study because it features all elements of a networked game, such as connections, transmissions, and even a bit of non-trivial sequential protocols.

We begin with the lobby data structure. The lobby contains only one field: the current players that are connected to the game:

```
world Lobby = {
  Players      : List<LobbyPlayer>
```

When a new instance of the game connects to the game, then all existing instances run once the rules inside their `master connect` scope:

```
  master {
    connect{
```

In our case, the existing instances will all reliably send the head of their players list, which contains the local player. They will then receive the new player that just connected, and add it to their local list of players:

```
      rule Players =
        send_reliable<LobbyPlayer>(Players.Head)
        let new_player = receive<LobbyPlayer>()
        yield Players + new_player
    }
  }
```

The new instance of the game that just connected to the game, runs once the rules inside its `slave connect` scope:

```
  slave {
    connect {
```

First the new instance receives all the `LobbyPlayer` instances that are currently initialized. A starting position which is not occupied by another entity is then calculated. Finally, the local entity is recreated with the new starting position, sent to the other instances of the game, and added to the list of other players in the game world:

```
      rule Players =
        let others = receive_many<List<LobbyPlayer>>()
        let max_x =
          maxby p in others
          select p.StartPosition.X
        let start_position = Vector2(max_x + 5.0f<pixel>, 0.0
           f<pixel>)
        let self = { Players.Head with Position =
           start_position }
        send_reliable<LobbyPlayer>(self)
        yield self + others
    }
  }
```

We wait until each player declares to be ready, and then we change the game world by switching from the lobby to the main arena:

```
  rule CurrentWorld =
    wait_until(forall p in Players
               select p.Ready)
    yield Arena.Create(Players.Head)
```

When the lobby is created, then it takes as input the string that contains the name of the local player and initializes the players list with only that player (put in the origin):

```
  Create(own_name) = {
    Players    = [Players.Create(own_name, Vector2.Zero)]
  }
}
```

The player entity contains the name of the player, a boolean which signals whether or not that player is ready to play or wishes to remain in the lobby a bit more, and the starting position:

```
entity LobbyPlayer = {
  Name            : string
  Ready           : bool
  StartPosition   : Vector2<pixel>
```

The entity performs a simple local computation: it waits until the user has pressed the `Enter` key, and then assigns to its own `Ready` field the `true` value. The entity also sends the value `true` to all of its own remote versions in other instances:

```
  master {
    rule Ready =
      wait_until(IsKeyDown(Enter))
      yield+send_reliable<bool> true
```

As a safeguard, we also force all players to automatically declare that they are ready after 30 seconds, in order to avoid players who keep others waiting for too long:

```
    rule Ready =
      wait(30.0f<s>)
      yield+send_reliable<bool> true
  }
```

The remote version of the entity in other game instances simply waits to receive its own `Ready` message, which it assigns locally:

```
  slave {
    rule Ready =
      yield receive<bool>()
  }
```

Finally, we create a lobby player from a name and an initial position. The player is initialized as not ready:

```
  Create(name, p) =
    {
      Name          = name
      Ready         = false
      StartPosition = p
    }
}
```

3.5.3 Game arena

In this section we discuss how a simple game arena can be defined with Casanova and its networking facilities. In the game we simply have a series of ships, one controlled by the player and others controlled by remote players. When a player shoots, he adds a projectile to his list of locally controlled projectiles. Projectiles are synchronized between instances, so that the projectiles created on an instance are *shown* (but have no real effect on the world) on the other game instances. When an instance of the game registers a hit of one of its *local* projectiles, then it locally adds a *hit* which it also synchronizes with the other instances. When an instance of the game receives a hit on its locally owned ship, then, it registers damage on itself.

In short, *local ships check for messages that tell them they have been hit*, while *hits are registered with the owner of the hitting projectile.*

The arena world contains the ship of the local player and those of the remote players:

```
world Arena = {
  Self          : Ship
  Others        : List<Ship>
```

When the instance connects to others, it sends its own local ship:

```
  master {
    connect {
      rule Self = send_reliable<Ship>(Self)
    }
  }
```

When the instance connects to others, it also receives their local ships and adds them to the `Others` list:

```
  slave {
    connect {
      rule Others = yield receive<Ship>() + Others
    }
  }
```

As an alternative, the rule above could be rewritten by using `receive_many` primitive. In this case, instead of adding ships to the game as soon as they connect, we block the system until a ship has been received from each connected instance. We could even avoid processing user input until the `Others` list is not empty, in order to start the game at the same time on all instances:

```
      rule Others = yield receive_many<List<Ship>>()
```

We create the local instance of the game from the `LobbyPlayer` that we defined in the previous section. We simply use the starting position to initialize the local ship, and start the instances of the other players as an empty list:

```
  Create(self:LobbyPlayer) =
    {
      Self    = Ship.Create(self.StartPosition)
      Others  = []
    }
}
```

The ship entity contains many fields. On one hand, it stores the position, velocity, and health:

```
entity Ship = {
  Position     : Vector2<pixel>
  Velocity     : Vector2<pixel/s>
  Health       : float<health>
```

The ship also contains the list of projectiles that it has shot so far:

```
  Shots        : List<Projectile>
```

Whenever another ship is hit by one of the local projectiles, then it is added to the list of `Hits`. The hits are synchronized so that other instances of the game can register damage to their local ships when it is hit by remote instances. Notice that we store the hit ships as `Ref`, because we just store a pointer to them, which Casanova will then skip when updating and drawing the various entities:

```
  Hits         : List<Ref<Ship>>
```

The local instance of a ship updates and sends the position and the velocity with `yield+send`, which at the same time updates the local value of a field and sends it across the network. Some input-specific code determines how the ship turns and changes direction of movement:

```
master {
  rule Position =
    yield+send<Vector2<pixel>> Position + Velocity * dt
  rule Velocity =
    ... input-specific code
    yield+send<Vector2<pixel>> ...
```

The ship also registers its updated health by subtracting the number of hits from others to itself:

```
  rule Health =
    yield+send<float<health>>(
      Health - from o in world.Others
               from h in o.Hits
               where h = Self
               select 1
               sum)
```

Whenever the ship registers a new shot, it sends it across the network and also stores it to the `Shots` list:

```
  rule Shots =
    ... input-specific code
    let new_shot = ...
    send<Projectile>(new_shot)
    yield new_shot + Shots
```

We split the current shots into two lists. On one hand, we keep the shots which have not yet hit any `Other` ship and we put them into `shots'`. On the other hand, we find all the hits that were hit by at least one shot and we put the into `hits'`. We reliably send these new hits across the network, because we need to communicate to other instances that they need to "damage themselves", and we must do so reliably because the hitting event is very important. We also locally keep the new hits and the new shots:

```
  rule Hits,Shots =
    ... partition Shots into shots and hits
    let hits',shots' = ...
    send_reliable<List<Hits>>(hits')
    yield hits', shots'
}
```

Remote instances of a ship receive all the values of updated position, velocity, and health:

```
  slave {
    rule Position = yield receive<Vector2<pixel>>()
    rule Velocity = yield receive<Vector2<pixel/s>>()
    rule Health = yield receive<float<health>>()
```

Whenever a new shot is received, then it is added to the local list:

```
    rule Shots =
      let new_shot = receive<Projectile>()
      yield new_shot + Shots
```

Whenever a new set of shots is received, we wait for a tick (so that they can be processed by reducing the health of the local ship as needed) and then remove them:

```
    rule Hits =
      yield receive<List<Hits>>()
      yield []
  }
```

Inactive shots are removed from the list of shots. This is done outside either the `master` or the `slave` blocks, and as such even remote instances of a ship will remove inactive projectiles:

```
  rule Shots =
    from s in Shots
    where s.Active
    select s
```

We create a ship from an initial position and with full health, no shots, and no hits:

```
  Create(p) =
    {
      Position = p
      Velocity = Vector2.Zero
      Health   = 100.0<health>
      Shots    = []
      Hits     = []
    }
}
```

The projectile, similarly to the ship, contains a position, a velocity, and an `Active` flag to determine when the projectile is to be removed:

```
entity Projectile = {
  Position    : Vector2<pixel>
  Velocity    : Vector2<pixel/s>
  Active      : bool
```

All projectiles, regardless of whether they are local or remote, update their position according to the usual physical rules:

```
  rule Position = Position + Velocity * dt
```

The local instance of a projectile sends, every few seconds, the position and the velocity to the remote instances:

```
  master {
    rule Position =
      wait 5.0<s>
      send<Vector2<pixel>>(Position)
    rule Velocity =
      wait 10.0<s>
      send<Vector2<pixel/s>>(Velocity)
```

After twenty seconds the projectile is registered both locally and remotely as inactive:

```
    rule Active =
      wait 20.0<s>
      yield+send_reliable<bool> false
  }
```

The remote versions receive the sent values:

```
  slave {
    rule Position =
      yield receive<Vector2<pixel>>()
    rule Velocity =
      yield receive<Vector2<pixel/s>>()
    rule Active =
      yield receive<bool>()
  }
```

Finally, we create a projectile from its position and velocity, and we set it to `Active`:

```
  Create(p,v) =
    {
      Position = p
      Velocity = v
      Active   = true
    }
}
```

3.5.4 Disconnection

Disconnection is not handled with explicit primitives. Rather, disconnection can be handled with a mixture of the networking primitives that we have seen so far and the default primitives that Casanova offers.

We add to the entity that handles disconnection a boolean flag and a time stamp. One flag is for disconnection, the other is for pings:

```
Disconnected : bool
LastPing     : Time
```

The local instance of an entity sends a ping every few seconds that signals that the entity is not disconnected. The ping is of type `Unit`, meaning that it contains no data whatsoever:

```
master {
  rule LastPing =
    wait 5.0<s>
    send<Unit>()
}
```

The remote instance reads the ping messages and resets the last ping time whenever it receives something:

```
slave {
  rule LastPing =
    receive<Unit>()
    yield Now
```

If no ping has been received within a reasonable time, then we set the `Disconnected` flag to true:

```
  rule Disconnected =
    wait (Now - LastPing > 30.0<s>)
    yield true
}
```

As a side note, when an instance is communicating with another one from *inside a rule*, within a complex protocol, then in case of disconnection between one of the parties the rule execution will be aborted midway. This is needed in order to prevent protocols stuck midway (after the other party disconnected) and stealing messages from other communications.

3.5.5 Verbose syntax

In some cases, the type of the message is ambiguous inside an entity. For example, when transferring multiple boolean flags, the language may be unable to understand when to receive one message or the other.

Consider an entity that contains two boolean fields, `A` and `B`:

```
A : bool
B : bool
```

The local instance of the entity updates A and B according to some internal logic, such as input, AI, etc.:

```
master {
  ... logic for updating A and B
```

Both A and B are sent across the network to the remote instances:

```
  rule A = yield+send<bool>(A)
  rule B = yield+send<bool>(B)
}
```

The remote instances try to receive A and B, but the compiler has no way to determine what the intention of a message was:

```
slave {
  rule A = yield receive<bool>()
  rule B = yield receive<bool>()
}
```

It may seem intuitively reasonable to match sends and receives depending on the rule name, but this can lead to a system which is too restrictive: it may be that the developer really wishes to swap A and B between the master and the slave.

A better solution is to give a compiler error for ambiguous cases, and at the same to offer explicit `send` and `receive` primitives with a user-defined *label*. Ambiguous communication operations will be matched depending on the label, and not only on the type:

```
send[ID]<T>(E:T) : Unit
receive[ID]<T>() : T
```

The example above would then become:

```
master {
  rule A = yield+send[A]<bool>(A)
  rule B = yield+send[B]<bool>(B)
}

slave {
  rule A = yield receive[A]<bool>()
  rule B = yield receive[B]<bool>()
}
```

Notice that we could also use labels which are not related to the names of the fields which are sent, for example to have more descriptive sources in the case where we swap the values of A and B [1]:

[1] Imagine that X and Y are descriptive labels that capture the essence of the communication.

```
master {
  rule A = yield+send[X]<bool>(A)
  rule B = yield+send[Y]<bool>(B)
}

slave {
  rule A = yield receive[Y]<bool>()
  rule B = yield receive[X]<bool>()
}
```

4 Syntax and semantics

4.1 Syntax

The syntax of networking in Casanova is rather simple. In the following we only provide an intuitive illustration of the terms that can be used, and a first description of their purpose.

The first series of supported keywords are those that are used for determining ownership of entities. The keywords below are all used to delimit the *scope* within which a given set of rules is valid:

```
master
slave
connect
```

Every entity in Casanova is duplicated across all the current instances of the game. Only one of this instances has ownership of the entity, that is acts as the authoritative instance which updates the entity for all other instances. The rules that perform such updates, and which also send the updates to the other instances, are all defined inside a `master` block. The rules that are executed in all the other, remote, instances are all defined inside a `slave` block.

When a new instance of the game connects, then we also run, just once, the rules inside the `connect` block. When a new instance is run, then its `master connect` rules are run once. When existing instances, on the other hand, witness the start of a new instance, then their `slave connect` rules are run once.

There are only four primitives for transmitting data across the network. Two are for sending data, and two are for receiving. Sending simply takes as input a value of any type, and returns nothing. Sending may also be done reliably, thereby trying to ensure that the other party has indeed received the message. Reliable sending may fail, for example if the receiver disconnects during the transmission. For this reason, sending reliably returns a boolean value that will be `true` if the transmission was successful, and `false` if the transmission failed for some reason:

```
send<T> : T -> Unit
send_reliable<T> -> bool
```

As a convenience, it is possible to, at the same time, locally store a value and send it across the network. For this purpose, we can use the `+send` syntax, which both *sends and returns* the expression that is passed as an argument to `+send`:

```
yield +send<T>(x)
```

Receiving may also be done in two different manners. Simple reception of a message is done with the `receive` primitive that returns a value of some type `T`. Receiving may also be done by all other instances at the same time, for example when voting or for other kinds of global synchronization. In this case, we wait until all other instances have each sent their value of some type `T`, and then we return all said values in a list of `T`s:

```
receive<T> : Unit -> T
receive_many<T> : Unit -> List<T>
```

4.2 Semantics

In the following we discuss the semantics of Casanova networking, but not in formal terms. Rather, we suggest a translation from Casanova with networking into Casanova without networking, assuming that one of the external libraries that Casanova is using is providing some low-level networking service.

Networking in Casanova is based on two separate systems. The first such system is the underlying networking library that is accessed as an external service to be orchestrated. This is directly *linked* to all programs that need networking functionalities. Multiple versions of this service may exist for different networking libraries, but in general we can assume that not many such services need to be built, and that there is a one-to-many relationship between networking services and actual games. The second system is the Casanova compiler itself, which modifies entity declarations and even defines whole new entities. The compiler will, effectively, translate away all networking operations and keywords (even `master`, `slave`, and `connect`) and turn them into much simpler operations on lists. The only assumption made that really has anything to do with networking is that some special memory locations are actually written to or read from the network.

4.2.1 Common primitives

We now present the common primitives that are provided by the networking service. The core of the networking service is the `NetManager`. The `NetManager` maintains the connections between the local instance of the game and the remote instances:

```
entity NetManager = {
```

The `NetManager` maintains a list of `NetPeers`. Each `NetPeer` represents a remote instance of the game. The `NetManager` also store the unique `Id` associated with the local instance:

```
  Peers          : List<NetPeer>
  Id             : PeerId
```

The `NetManager` also manages two flags, which will be used to determine when the `master connect` and the `slave connect` rules are run:

```
  MasterConnect : bool
  SlaveConnect  : bool
```

The `MasterConnect` flag may run only once, at the beginning of the game. The game world will then reset the flag to `false` when the `master connect` rules are all terminated.

Whenever a new connection is established with a new `NetPeer`, then we add that peer to the list of `Peers`, and we set `SlaveConnect` to `true` so that the local `slave connect` rules may be run. Notice that we wait for `SlaveConnect` to be set to false, which is only done by the game world when the current `slave connect` rules all terminate. This allows us to process all new connections one at a time:

```
  rule Peers, SlaveConnect =
    wait_until(SlaveConnect = false)
    let new_peer = wait_some(NetPeer.NewPeer())
    yield Peers + new_peer, true
```

Every few seconds, we check which peers disconnected and remove them from the list of peers. The disconnection is all managed internally by the underlying networking library:

```
  rule Peers =
    wait 5.0f<s>
    from p in Peers
    where p.Channel.Connected
    select p
```

We initialize the `NetManager` by finding all reachable peers across the network. We use their current `Id` values to find an `Id` for this instance that is unique to this game session. We also set `MasterConnect` to true, since we need to send the locally managed values to the other instances, and `SlaveConnect` to false:

```
  Create() =
    let peers = NetChannel.FindPeers()
    {
      Id            = from p in peers
                        max_by p.Id + 1
      Peers         = peers
      MasterConnect = true
      SlaveConnect  = false
    }
}
```

Another, remote instance of the game is represented by the `NetPeer`. A `NetPeer` is responsible for handling the actual communication to the other instances of the game:

```
entity NetPeer = {
```

The `NetPeer` contains a channel, which is an instance of a data-type supplied by some network library and which will, automatically, send and receive messages. The `NetPeer` also contains an `Id` which uniquely identifies it among the various instances of the game, and a list of messages received so far:

```
  Channel            : NetChannel
  Id                 : PeerId
  ReceivedMessages   : List<InMessage>
```

The list of messages received so far is constantly refreshed with the list of received messages automatically populated by the channel:

```
  rule ReceivedMessages = yield Channel.ReceivedMessages
```

The `NetPeer` also looks for all the messages that need to be sent across the game world, both reliably and unreliably. These messages are then written into the `SentMessages` and `ReliablySentMessages` lists of the underlying channel:

```
  rule Channel.SentMessages =
    from (m:OutMessage) in *
    where exists(Id, m.Targets) || m.Targets = []
    select m
  rule Channel.ReliablySentMessages =
    from (m:ReliableOutMessage) in *
    where exists(Id, m.Targets) || m.Targets = []
    select m
```

```
}
```

The underlying networking library is also expected to provide a series of data types which represent messages and channels. We do not care about the concrete shape of the data types, as long as they contain the required properties.

The simple message only needs to handle the *Casanova header*, which stores which instance of the game sent this message, what type of data the message contains, and the entity from which this message was sent:

```
interface Message
  Sender          : PeerId
  ContentType     : TypeId
  OwnerEntity     : EntityId
```

An outgoing message inherits from `Message`. It also has a list of target instances to which this message is addressed. The list of targets may also be empty, in which case we wish to send the message to all reachable peers. An `OutMessage` also offers a series of low-level write methods to send various primitive values such as integers, floating-point numbers, strings, etc.:

```
interface OutMessage
  inherit Message
  Targets             : List<PeerId>
  member WriteInt     : int -> Unit
  member WriteFloat   : float32 -> Unit
  member WriteString  : string -> Unit
  member WriteT       : T -> Unit // only for elementary data
     -types
```

Almost identical to the `OutMessage` is the `ReliableOutMessage`. A reliable outgoing message only differs from a simple outgoing message in that it also has properties that tells us whether or not the message has been received or the transmission has failed:

```
interface ReliableOutMessage
  inherit Message
  Targets             : List<PeerId>
  member WriteInt     : int -> Unit
  member WriteFloat   : float32 -> Unit
  member WriteString  : string -> Unit
  ...
  member Received     : bool
  member Failed       : bool
```

A received message inherits from the simple `Message`, and also offers a series of low-level write methods to read various primitive values such as integers, floating-point numbers, strings, etc.:

```
interface InMessage
  inherit Message
  member ReadInt      : Unit -> int
  member ReadFloat    : Unit -> float32
  member ReadString   : Unit -> string
  ...
```

The final data-type that is provided by the networking library is the communication channel itself. Casanova requires a channel to expose the messages which were just received, and lists where the messages to be sent can be put. Also, the channel should provide a (static) mechanism to find those peers that just connected:

```
 interface NetChannel
  member ReceivedMessages        : List<InMessage>
  member SentMessages            : List<OutMessage>
  member ReliablySentMessages    : List<ReliableOutMessage>
  static member FindPeers        : Unit -> List<NetPeer>
```

Notice that in the listings above we have slightly abused the notion of *interface*. We have used a notion that resembles more closely that of a *type-trait* or a *type-class*, but the abuse is quite minor and we believe the idea of an interface to capture the essence of what Casanova expects from the underlying library.

4.2.2 Chat sample translated

Inside an application, the compiler generates a series of additional entities and modifies the game rules in order to accommodate networking primitives. The generated entities are all wrappers for messages, both incoming and outgoing. A pair of incoming/outgoing message entities is created for each type `T` such that a `send<T>` and a `receive<T>` appear in the game rules. In the case of the chat sample, we only ever send strings, so only one such pair is generated.

One generated entity inherits from `InMessage` and contains a string value which was just received:

```
entity InMessageString = {
  inherit InMessage
  Value : string
```

When creating an `InMessageString`, we take a simpler `InMessage`, "parse"[2] it by invoking `ReadString` once, and then store message and string:

[2] *Parsing* in this context is a bit of an excess.

```
  Create(msg : InMessage) =
    let value = msg.ReadString()
    {
      InMessage = msg
      Value     = value
    }
}
```

The dual of the entity we have just seen is the `ReliableOutMessageString`, which inherits from the simpler `ReliableOutMessage`:

```
entity ReliableOutMessageString = {
  inherit ReliableOutMessage
```

When we create a `ReliableOutMessageString`, in reality we create a `ReliableOutMessage` and write the content of the message (a string) to it with `WriteString`:

```
  Create(value : string, targets : List<Peer>, sender :
      ConnectionId, owner_entity : EntityId) =
    let m = new ReliableOutMessage(sender, StringTypeId,
        owner_entity, targets)
    do m.WriteString(m)
    {
      ReliableOutMessage = m
    }
}
```

At this point we can move on to the definition of the game world. The first two fields are identical to the sample as we have seen it in previously. Local rules that do not `send` or `receive` are unchanged:

```
world Chat = {
  Text                  : string
  Line                  : string

  ...
```

The compiler also adds a series of additional fields. One of the fields is a network manager, which will manage the various connections. Two lists, one for incoming and one for outgoing messages are also declared. Finally, an `Id` is used to store a unique identifier for this specific entity:

```
  Network               : NetManager
  Inbox                 : List<InMessageString>
  Outbox                : List<ReliableOutMessage>
  Id                    : EntityId
```

We automatically empty the list of outgoing messages, in the assumption that the `NetPeer` instances have already stored them and are handling them:

```
rule Outbox = yield []
```

We fill the list of incoming messages from the incoming messages found in the channels of the various peers. We filter those messages, so that only those that were specifically aimed towards this entity (and contain data of the expected type, in our case `string`) are kept:

```
rule Inbox =
  yield from c in Network.Peers
        from m in c.ReceivedMessages
        where m.ContentType = StringTypeId &&
              m.OwnerEntity = Id
        select InMessageString.Create(m.Value)
```

When we want to send a string, we also create a message with the string we wish to send, add it to the `Outbox` list, and wait until the message is received. The rest of the rule is unchanged:

```
rule Line, Text, Outbox =
  wait_until(IsKeyDown(Keys.Enter))
  let msg = ReliableOutMessageString.Create(Line, [],
     Network.Id, Chat.TypeId, Id).ReliableOutMessage
  yield Outbox <- Outbox + msg
  wait_until(msg.Received)
  yield Line <- "", Text <- Text + "\n" + Line
```

Essentially, `send_reliably<string>` turns into the following lines:

```
  let msg = ReliableOutMessageString.Create(Line, [],
     Network.Id, Id).ReliableOutMessage
  yield Outbox <- Outbox + msg
  wait_until(msg.Received)
```

In particular, we create the output message by also specifying:

- the message recipients, which are the empty list `[]` which means that the message will be sent to all other peers
- the peer that is the sender (and owner) of the message, which is `Network.Id`
- the entity that the message was sent from, which is simply the world `Id`

We consider a message received when it appears in the `Inbox` list. When we find one, we remove it from the list, and process it as the result of the `receive` function:

```
  rule Text, Inbox =
    wait_until (Inbox.Length > 1)
    yield Text + "\n" + Inbox.Head.Value, Inbox.Tail
```

Essentially, `receive<string>` has turned into:

```
    wait_until (Inbox.Length > 1)
    Inbox.Head.Value // the received string
```

The creation of the world now simply initializes the network manager, creates a new unique id for the entity, and then initializes the various other fields with empty values:

```
  Create() =
    let network = NetManager.Create()
    let id = NetManager.NextId()
    {
      Text                 = ""
      Line                 = ""
      Id                   = id
      Network              = network
      Inbox                = []
      Outbox               = []
      StringsInbox         = []
    }
}
```

As we can see from the sample, it is possible to translate away the networking primitives, provided a very small component capable of sending and receiving messages created from an aggregation of elementary data structures.

4.2.3 Lobby sample translated

[[[Tears, sweat and blood go in here.]]]

5 Evaluation

Building a small game or various fragments of existing games. Transmission benchmarks. Performance overhead. Reliability.

6 Conclusions

Adding multi-player support to games is a highly desirable feature. By letting players interact with each other, new forms of gameplay, cooperation, and competition emerge without requiring any additional effort by the developer. This allows a game to remain fresh and playable, even after the single player content has been exhausted.

Multi-player support in games is a very expensive piece of software to build. Low-level protocols need to be programmed with both performance and reliability in mind, and existing abstraction mechanisms such as SOAP, RPC, etc. usually fall short from various points of view.

In this paper we have presented a model for networking especially geared towards multi-player games. This model allows the creation of robust, high-performance multi-player games based on a peer-to-peer architecture. We have shown the approach to be flexible enough to build some small game-like applications, and we have given some preliminary evidence that performance is good acceptable for real-time games both in terms of computational time required and bandwidth used.