Communicating parallel games

M. Abbadi A. Cortesi F. Di Giacomo A. Plaat
P. Spronck D. Dini G. Costantini G. Maggiore

May 24, 2014

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 design of game mechanics. This allows a game to remain fresh and playable, even after the single player content has been exhausted. For example, consider any modern AAA game such as *Halo 4*. After months since its initial release, most players have exhausted the single player, narrative-driven campaign. Nevertheless the game remains heavily in use thanks to multiplayer modes, which in effect extended the life of the game significantly. This phenomenon is even more evident with games such as *World of Warcraft* or *EVE*, where multiplayer is the only modality of play and there is no single-player experience. Nowadays essentially all AAA games feature some form of multiplayer. On the other hand, unfortunately, smaller games such as casual games, serious games, mobile games, etc. very often lack multiplayer. We believe this phenomenon to be caused by technical issues.

Challenges Multi-player support in games is a very expensive piece of software to build. Multiplayer games are under strong pressure to have very good *performance*. Performance is both in terms of CPU time, and in bandwidth used. Also, games need to be very *robust* with respect to transmission delays, packets lost, or even clients disconnected. 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"), or to try to abuse the game and its technical flaws to gain advantages or to disrupt the experience of others.

Networking code reuse is quite low across titles and projects. This comes from the fact that 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, previous effort is substantially inaccessible for new titles.

Existing approaches Networking in games is usually built with either very low level or very high level mechanisms. Very low level mechanisms are based on manually sending streams of bytes and serializing only the essential bits of the game world, usually incrementally, on unreliable channels (UDP). This coding process 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. Low-level mechanisms must also be very robust, making the task even harder.

High level protocols such as RDP, reflection-based serialization, etc. can also be used. These methods greatly simplify networking code, but are rarely used in complex games and scenarios. The requirements of performance mean that many high-level protocols or mechanisms are insufficient, either because they are too slow computationally (especially when the rely on reflection) or because they transmit too much data across the network.

Our approach To avoid the problems of both existing approaches, we propose a middle ground. 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. Until now, common primitives have not been syntactically and semantically captured inside existing languages. Using the right level of abstraction, these general patterns of networking can be captured, while leaving full customization power in the hand of the developer (to apply such primitives to any kind of game).

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.

Finally, we observe that a good system for building multiplayer in games should support timed-protocols and in general a complex flow of control that explicitly mentions time. For example, concurrent waiting operators should be a first class element, in order to be able to express a concept such as *perform an operation after you receive a specific message*.

Contribution of this work. In this paper we present a programming language with explicit support for networking in games. Our technique is a middle ground between high level and low level networking solutions, and it allows building networking protocols with the desired level of granularity and performance.

Our approach. We begin by discussing networking in games and the technical challenges it presents (Section 2). We then provide the general intuition of our system and its primitives, and we show how to build some example games in it (Section 3). We then formalize the system by providing the language syntax and semantics (Section 4). Finally, we provide an evaluation of the language based on a user study and a series of performance and bandwidth benchmarks (Section 5).

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. The evaluation, which is discussed

in Section 5, will be a mixture of analytical discussion, user studies, and measured benchmarks.

2.1 Resource usage

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.

Similarly, a game needs to be able to present a new, updated picture on the screen at a rate (between 30 and 60 times per second) 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.

2.2 Reliability

Messages do not always arrive at destination. This may be a temporary issue, for example when some messages are simply lost during transmission. Also, 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.

2.3 Expressive power

A system designed for building multi-player games should, first and foremost, be expressive enough to cover the definition of various games. Moreover, its semantics should be *appropriate*, meaning that the various idioms of networked communication in games should be easy to express without abusing

the available primitives. As such, a mechanism for dealing with networking should offer a simplified view of networking primitives. Such primitives need to be:

- *few*, because we wish to capture the essential nature of the problem
- *orthogonal*, because we wish the primitives to have no semantic overlap: each primitive cannot be replaced by the others
- *intuitive*, because learning to use the system should not require deep understanding through a steep learning curve

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 *local* of the entity. The instances of the game that received that entity across the net, on the other hand, are called the *remotes* of the entity. This means that each entity can have two sets of rules: one for its local instance, and one for its remote instances. The local 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 remotes. The remote rules will usually (but not necessarily) receives the updates from the local 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. *local connections* will usually send the locally owned entity to the new instance. *remote 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.

Finally, we take advantage of type inference. This means that, even though we could write `send<T>(x)`, the type inference engine automatically determines, from the parameter `x` itself, the type `T`. This allows us to write the much simpler `send(x)` (the same applies to all other networking primitives).

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
  Keyboard            : Keyboard
```

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 -> 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(from c in Line
               exists_by c = '\n')
    send_reliable(Line)
    yield "", Text + Line
```

Whenever a character is pressed, then we add it to the line we are writing:

```
  rule Keyboard -> Line =
    wait_until (Keyboard.PressedChars <> [])
    yield Line + Keyboard.PressedChars
```

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 -> Text =
    yield Text + receive()
```

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     = ""
      Keyboard = Keyboard.Create()
    }
}
```

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 one field for the local player and a list of other (remote) players that are connected to the game:

```
world Lobby = {
  Self     : LobbyPlayer
  Others   : List<LobbyPlayer>
```

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

```
local {
  connect{
```

In our case, the existing instances will all reliably send their local players, which are all received with `receive_many`. Those players will be stored in the `Others` list, and used to find a free position for the local player. The local player is then re-created with the new position and sent to the other instances:

```
    rule Self, Others -> Self, Others =
      let others = receive_many()
      let max_x =
        maxby p in others
        select p.StartPosition.X
      let self = { Self
                   with Position.X = max_x + 5.0f<pixel> }
      yield +send_reliable(self), others
  }
}
```

The other instances of the game simply do the opposite operation upon connection of a new player, inside the `remote connect` block.

```
remote {
  connect {
```

First the new instance (reliably) sends its own local player. We receive the new player instance which was just computed remotely and store it locally in the `Others` list:

```
    rule Self,Others -> Others =
      send_reliable(Self)
      let new_player = receive()
      yield Others + new_player
  }
}
```

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 Self,Others -> CurrentWorld =
  wait_until(Self.Ready)
  wait_until(forall p in Others
             select p.Ready)
  yield Arena.Create(Self)
```

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

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>
  Keyboard         : Keyboard
```

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:

```
  local {
    rule Keyboard, Ready =
      wait_until Keyboard.EnterPressed
      yield+send_reliable 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 true
  }
```

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

```
  remote {
    rule () -> Ready =
      yield receive()
  }
```

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 reliably sends its own local ship and receives the ships of the other instances:

```
  local {
    connect {
      rule Self -> Others =
        send_reliable(Self)
        yield receive_many()
    }
  }
```

When the instance connects to others, the others receive its ships and add them to the `Others` list, and send their own local ship:

```
  remote {
    connect {
```

```
      rule Self,Others -> Others =
        let new_other = receive()
        send_reliable(Self)
        yield Others + new_other
    }
  }
```

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:

```
  local {
    rule Position,Velocity -> Position =
      yield+send Position + Velocity * dt
    rule ... -> Velocity =
```

```
    ... input-specific code
    yield+send ...
```

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

```
  rule World.Others, Health -> Health =
    yield+send(
      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, ... -> Shots =
    ... input-specific code
    let new_shot = ...
    send(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 -> Hits,Shots =
    ... partition Shots into shots and hits
    let hits',shots' = ...
    send_reliable(hits')
    yield hits', shots'
}
```

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

```
remote {
  rule () -> Position = yield receive()
  rule () -> Velocity = yield receive()
  rule () -> Health = yield receive()
```

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

```
  rule Shots -> Shots =
    let new_shot = receive()
```

```
      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()
      yield []
  }
```

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

```
  rule Shots -> 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,Velocity -> Position =
    yield Position + Velocity * dt
```

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

```
  local {
    rule Position -> Position =
      wait 5.0<s>
      send(Position)
    rule Velocity -> Velocity =
      wait 10.0<s>
      send(Velocity)
```

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

```
    rule () -> Active =
      wait 20.0<s>
      yield+send_reliable false
  }
```

The remote versions receive the sent values:

```
  remote {
    rule () -> Position =
      yield receive()
    rule () -> Velocity =
      yield receive()
    rule () -> Active =
      yield receive()
  }
```

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:

```
local {
  rule () -> LastPing =
    wait 5.0<s>
    send()
}
```

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

```
remote {
  rule () -> LastPing =
    receive()
    yield Now()
```

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

```
  rule LastPing -> 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.:

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

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

```
  rule A -> A = yield+send(A)
  rule B -> B = yield+send(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:

```
remote {
  rule () -> A = yield receive()
  rule () -> B = yield receive()
}
```

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 local and the remote.

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:

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

remote {
  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]:

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

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

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

A more complex motivation Sending and receiving with an explicit ID for the operation is an advanced feature that is most often needed in complex scenarios. For this reason it is relatively hard to come up with a convincing, yet simple, example that uses it. It is possible, on the other hand, to describe the abstract class of game rules that may benefit from this kind of computation.

Suppose that we have two fields in an entity. Field `A` is a boolean, and as such can be transmitted easily across instances with minimal bandwidth usage. Field `X`, on the other hand, has type `T`, which we assume to be *too large* to transmit often across the network:

```
A : bool
X : T
```

There are a series of similar rules that all compute a new value for `A`, and use that value to compute a new value for `X`. Each rule uses a different way to compute both `A` and `X`:

```
local {
  rule A,X -> A,X =
    let A' = a1()
    let X' = f1(A')
    yield +send(A'), X'
  rule A,X -> A,X =
    let A' = a2()
    let X' = f2(A')
    yield +send(A'), X'
  ...
  rule A,X -> A,X =
    let A' = aN()
    let X' = fN(A')
    yield +send(A'), X'
}
```

When we receive a value of `A`, we can compute the correct value of `X` without having to transmit it:

```
remote {
  rule X -> A,X =
    let A' = receive()
    let X' = f1(A')
    yield A', X'
```

```
  rule X -> A,X =
    let A' = receive()
    let X' = f2(A')
    yield A', X'
  ...
  rule X -> A,X =
    let A' = receive()
    let X' = fN(A')
    yield A', X'
}
```

Unfortunately, the system has no way of determining which `send` is paired with which `receive`. For this reason we will have to tag each transmission with an appropriate ID to disambiguate:

```
local {
  rule X -> A,X =
    let A' = a1()
    let X' = f1(A')
    yield +send[ID1](A'), X'
  rule X -> A,X =
    let A' = a2()
    let X' = f2(A')
    yield +send[ID2](A'), X'
  ...
}

remote {
  rule X -> A,X =
    let A' = receive[ID1]()
    let X' = f1(A')
    yield A', X'
  rule X -> A,X =
    let A' = receive[ID2]()
    let X' = f2(A')
    yield A', X'
  ...
}
```

Thanks to this disambiguation the system now knows that the various `send` operations will only feed data in the appropriate `receive` slots.

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:

```
local { ... }
remote { ... }
```

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 `local` block. The rules that are executed in all the other, remote, instances are all defined inside a `remote` block.

Another keyword, which is used nested in `local` or `remote`, is:

```
connect { ... }
```

When a new instance of the game connects, then we also run, just once, all the rules inside the `connect` block. When a new instance is run, then its `local connect` rules are run once. When existing instances, on the other hand, witness the start of a new instance, then their `remote 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 `local`, `remote`, 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 `NetPeer`s. 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 `local connect` and the `remote connect` rules are run:

```
  localConnect : bool
  remoteConnect  : bool
```

The `localConnect` flag may run only once, at the beginning of the game. The game world will then reset the flag to `false` when the `local 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 `remoteConnect` to `true` so that the local `remote connect` rules may be run. Notice that we wait for `remoteConnect` to be set to false, which is only done by the game world when the current `remote connect` rules all terminate. This allows us to process all new connections one at a time:

```
  rule Peers, remoteConnect =
    wait_until(remoteConnect = 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 `localConnect` to true, since we need to send the locally managed values to the other instances, and `remoteConnect` to false:

```
  Create() =
```

```
    let peers = NetChannel.FindPeers()
    {
      Id            = from p in peers
                      max_by p.Id + 1
      Peers         = peers
      localConnect = true
      remoteConnect  = 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:

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

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

```
      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:

```
    yield Inbox +
          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

In this more complete example we also see how local and remote blocks are handled, both at connection time and during the main run.

The lobby sample generates four entities for (reliably) sending and receiving `bool` and `LobbyPlayer` values. The `bool` message entities are almost identical to the `string` ones that we have seen in the chat sample, and so we omit them. The `LobbyPlayer` message entities, on the other hand, are more articulated. A received `LobbyPlayer` will have an underlying incoming message and the `LobbyPlayer` itself:

```
entity InMessageLobbyPlayer = {
  inherit InMessage
  Value : LobbyPlayer
```

When we create the `InMessageLobbyPlayer`, we parse its contents. First we read the player `name`, then its `ready` status, and then the `X` and `Y` of its starting position. The `id` of the player is stored in the underlying message, so we do not need to read it again and can reuse it directly. Finally, the

owner of the entity is its sender, and the received entity will thus be remoted to the underlying message sender:

```
  Create(msg : InMessage) =
    let name         = msg.ReadString()
    let ready        = msg.ReadBool()
    let start_pos    = Vector2(msg.ReadFloat(), msg.ReadFloat
       ())
    let id           = msg.EntityId
    let ownership    = remote(msg.Sender)
    {
      InMessage = msg
      Value =
        {
          Name           = name
          Ready          = ready
          StartPosition  = start_pos
          Id             = id
          InboxBool      = []
          Outbox         = []
          Ownership      = ownership
        }
    }
}
```

Similarly, we define the `ReliableOutMessageLobbyPlayer` as a wrapper over the simpler `ReliableOutMessage`:

```
entity ReliableOutMessageLobbyPlayer = {
  inherit ReliableOutMessage
```

When we send a `LobbyPlayer` we first create the underlying message with the Casanova header of sender, owner entity, etc. Then, we perform a series of `write` operations that mirror the `read` operations in the `InMessageLobbyPlayer`:

```
  Create(value : LobbyPlayer, targets : List<Peer>, sender :
     ConnectionId, owner_entity : EntityId) =
    let m = new ReliableOutMessage(sender, LobbyPlayerTypeId,
        owner_entity, targets)
    do m.WriteString(value.Name)
    do m.WriteBool(value.Ready)
    do m.WriteFloat(value.StartPosition.X)
    do m.WriteFloat(value.StartPosition.Y)
    {
      ReliableOutMessage = m
    }
}
```

The `Lobby` itself contains the same fields that store the various players:

```
world Lobby = {
  Self                    : LobbyPlayer
  Others                  : List<LobbyPlayer>
```

Additionally, the compiler generates a series of networking-related fields. The entity has a networking manager, and `id`, and a series of lists for storing incoming and outgoing messages. In particular, it may seem as we receive `LobbyPlayers` twice, but we actually store the received lobby players for both `receive` and `receive_many`:

```
  Network                 : NetManager
  Id                      : int
  InboxLobbyPlayer        : List<InMessageLobbyPlayer>
  InboxListLobbyPlayer    : List<InMessageLobbyPlayer>
  Outbox                  : List<ReliableOutMessage>
```

Just like we did for the chat, we empty the list of outgoing messages, and fill in the lists of `LobbyPlayer` messages from the various peers:

```
  rule Outbox = yield []
  rule InboxListLobbyPlayer =
    yield InboxListLobbyPlayer +
          from c in Network.Peers
          from m in c.ReceivedMessages
          where m.ContentType = LobbyPlayerTypeId &&
                m.OwnerEntity = Id
          select InMessageLobbyPlayer.Create(m.Value)
  rule InboxLobbyPlayer =
    yield InboxLobbyPlayer +
          from c in Network.Peers
          from m in c.ReceivedMessages
          where m.ContentType = LobbyPlayerTypeIdTypeId &&
                m.OwnerEntity = Id
          select InMessageLobbyPlayer.Create(m.Value)
```

The `local connect` waits until the network manager sets its `localConnect` flag to `true`. This will only allow the rule to run once upon first connection, and then stop:

```
  rule Self, Others, InboxListLobbyPlayer, Outbox, Network.
     localConnect =
    wait_until(Network.localConnect = true)
```

We now perform a `receive_many` by waiting until all peers have sent us something. We then take one received `LobbyPlayer` per peer:

```
    wait_until(from m in InboxListLobbyPlayer
               select m
               group_by m.Sender
```

```
            count = Network.Peers.Length)
  let others = // take one item per peer
    from m in InboxListLobbyPlayer
    select m
    group_by m.Sender as g
    select g.Elements.Head.Value
  yield InboxListLobbyPlayer <- []
```

We then send our own `LobbyPlayer`, and wait until it has been received by all:

```
  let max_x =
    maxby p in others
    select p.StartPosition.X
  let start_position = Vector2(max_x + 5.0f<pixel>, 0.0f<
     pixel>)
  let self = { Self with Position = start_position }
  let msg = ReliableOutMessageLobbyPlayer.Create(Self, [],
     Network.Id, Id).ReliableOutMessage
  yield Outbox <- Outbox + msg
  wait_until(msg.Received)
```

Finally, we store the players locally, and reset `localConnect` to `false`. A very important notice is that, in case of multiple `local connect` rules, then we need to wait until all of them are done before resetting `localConnect`. This will require additional boolean flags:

```
  yield Self <- self, Others <- others, Network.
     localConnect <- false
```

The `remote connect` waits until the network manager sets its `remoteConnect` flag to `true`. This will only allow the rule to run once for every new connection, and then stop:

```
rule Others, InboxLobbyPlayer, Outbox, Network.
   remoteConnect =
  wait_until(Network.remoteConnect = true)
```

We send the local player to the new instance, and wait for the message to be received:

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

We then received the local player of the new instance:

```
  let new_player = receive<LobbyPlayer>()
  wait_until(InboxLobbyPlayer.Length > 1)
```

```
    let others = InboxLobbyPlayer.Head.Value
    yield InboxLobbyPlayer <- InboxLobbyPlayer.Tail
```

Finally, we add the new player to the list of players, and reset `remoteConnect` to `false`. A very important notice is that, in case of multiple `remote connect` rules, then we need to wait until all of them are done before resetting `remoteConnect`. This will require additional boolean flags:

```
    yield Others <- Others + new_player, Network.
       remoteConnect <- false
```

Starting the game waits for all players to be ready. This rule is unchanged:

```
  rule CurrentWorld =
    wait_until(Self.Ready &&
               from p in Others
               select p.Ready)
    yield Arena.Create(Self)
```

We create the game like we did for the chat sample. We initialize the network and local fields, and start the various inbox and outbox lists to empty lists:

```
  Create(own_name) =
    let network = NetManager.Create()
    let id = network.NextId
    {
      Self                  = LobbyPlayer.Create(own_name,
         Vector2.Zero, network)
      Others                = []
      Network               = network
      Id                    = id
      InboxLobbyPlayer      = []
      InboxListLobbyPlayer  = []
      Outbox                = []
    }
}
```

We can now present the `LobbyPlayer` itself. The entity contains its original fields of name, readiness, and initial position for the game:

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

The entity also has a network `id`, plus an `ownership` value that indicates whether or not the entity is owned by the local instance (in which case `Ownership = local`) or whether it is owned by a remote peer (in which case

`Ownership = remote(Peer)` where `Peer` is the `Id` of the owner). The entity also contains a list of received and unprocessed `bool` messages, and a list of outgoing messages that will be sent:

```
Id                      : int
Ownership               : NetOwnership
InboxBool               : List<InMessageBool>
Outbox                  : List<ReliableOutMessage>
```

We automatically empty the list of outgoing messages, in the assumption that the `NetPeer` instances have already stored them and are handling them, and we store locally the received messages of type `bool` and destined to this entity:

```
rule Outbox = yield []
rule InboxBool =
  yield InboxBool +
        from c in Network.Peers
        from m in c.ReceivedMessages
        where m.ContentType = BoolTypeId &&
              m.OwnerEntity = Id
        select InMessageBool.Create(m.Value)
```

The local rules all run exclusively if the entity is locally owned. For this reason we wait, just once, that `Ownership = local`, and then we loop forever the body of the rule because ownership does not change:

```
rule Ready, Outbox =
  wait_until(Ownership = local)
  while(true)
```

Whenever the `Enter` key is pressed, we create a reliable outgoing boolean message and then put it in the outgoing queue of messages. We then wait until the message is received:

```
    wait_until(IsKeyDown(Enter))
    let msg = ReliableOutMessageBool.Create(true, [], world
       .Network.Id, Id).ReliableOutMessage
    yield Outbox <- Outbox + msg
    wait_until(msg.Received)
```

Finally, we set the local value of `Ready` to `true`:

```
    yield Ready <- true
```

The remote rules all run exclusively if the entity is remotely owned. For this reason we wait, just once, that `Ownership <> local`, and then we loop forever the body of the rule because ownership does not change:

```
  rule Ready, InboxBool =
    wait_until(Ownership <> local)
    while(true)
```

We wait until a boolean message appears in the incoming queue. We then return this message, and discard it from the queue as it has now been processed:

```
      wait_until(InboxBool.Length > 1)
      yield InboxBool.Head.Value, InboxBool.Tail
```

We create the entity from its original parameters, but we also need the local `NetManager` in order to obtain a unique `Id` for the entity. We also initialize the various message lists to empty lists. Finally, when creating an entity locally we will always set its `Ownership` to `local`, because the entity is owned by the peer that creates it:

```
  Create(name, p, network : NetManager) =
    {
      Name          = name
      Ready         = false
      StartPosition = p
      Id            = network.NextId
      InboxBool     = []
      Outbox        = []
      Ownership     = local
    }
}
```

4.3 Formal semantics

We now present the semantics in a more formal and compact framework. To describe the way multi-player games work, we consider the various instances of the game running in lock-step. Each instance has its own game world:

$$\omega_1, \omega_2, \ldots$$

The game world is structured like a tree of entities. Each entity has some fields and some rules. Each rule acts on a subset of the fields of the entity by defining their new value after one (or more) ticks of the simulation. For simplicity, in the following we assume that each rule updates all fields together[3]:

[3] `rule X = yield 10` is equivalent to `rule X,Y,Z = yield 10,Y,Z`

$$E = f_1 \dots f_n \quad r_1 \dots r_m$$

An entity is updated by evaluating, in order, all the rules to the fields:

$$\begin{aligned} \text{tick}(e : E, dt) &= \text{tick}(f'_1, dt) \dots \text{tick}(f'_n, dt) \quad r'_1 \dots r'_m \\ f'_1, \dots, f'_n, r'_1, \dots, r'_m &= \text{step}(\dots \text{step}(f_1, \dots, f_n, r_1), \dots, r_m) \end{aligned}$$

We define the step function as a function that recursively evaluates the body of a rule. The function evaluates until it encounters either a wait or a yield statement. step also returns the remainder of the rule body, so that the rule will effectively be resumed where it left off, at the next evaluation of step:

$$\begin{aligned} \text{step}(f_1, \dots, f_n, \text{let}\, x = y \,\text{in}\, z) &= \text{step}(f_1, \dots, f_n, z[x := y]) \\ &\vdots \\ \text{step}(f_1, \dots, f_n, \text{yield}\, x; b) &= x, b \end{aligned}$$

In order to add networking, we assume that each entity has two new fields, which are id and owner. id is a simple (numeric) identifier that, inside a single instance of the game world, uniquely denotes a specific entity. owner may be either local or remote. Given a set of entities (one per game world) that share the same type and the same id, exactly one of them will be local, all the others will be remote.

First of all we "translate away" the local and remote scopes. This means that given an entity with some rules defined inside such scopes:

$$E = f_1 \dots f_n \quad r_1 \dots \quad \text{local}\, r_j \dots \text{remote}\, r_l \dots$$

we transform the rules r_j inside the local scope into:

$$\text{wait}(\text{owner} = \text{local}); r_j$$

and we transform the rules r_j inside the remote scope into:

$$\text{wait}(\text{owner} = \text{remote}); r_l$$

This will prevent those rules from running when they should not. Similarly, by adding the global localconnect and remoteconnect flags, we can have some rules only execute when a new instance is added to the game.

All existing instances will enable remoteconnect, while the new instance will enable localconnect.

When evaluating a rule with the step function, we stop at receive[T] statements. Assume that we are updating entities inside a specific game instance, the one with the world ω_i. We look for some entity, e', belonging to *another* game world, with the same id as the entity we are updating, and which is sending a value v of the type we are expecting (T):

$$\text{step}(f_1, \ldots, \text{let } x = \text{receive}[T](); b) = \begin{cases} f_1, \ldots, b[x := v'] & \text{when } \exists r_l \in e' : r_l = \text{send}[T](v) \\ f_1, \ldots, \text{let } x = \text{receive}[T](); b & \text{otherwise} \end{cases}$$

Notice that when we receive a value v, we turn it into v' in the receiving instance of the game. v' is identical to v, but all of its owner fields are changed to remote. This is needed in order to reflect the fact that even if the transmitted entity was owned by the sender, it is certainly not owned by the receiver. Similarly, when performing a receive_many, then we look for another entity e'_j with the same type and id of the current entity *for each other game world*:

$$\text{step}(f_1, \ldots, \text{let } x = \text{receive\_many}[T](); b) = \begin{cases} f_1, \ldots, b[x := [v'_1 \ldots]] & \text{when } \forall e'_j : \exists r_l^j \in e'_j : r_l^j = \text{ser} \\ f_1, \ldots, \text{let } x = \text{receive\_many}[T](); b & \text{otherwis} \end{cases}$$

receive_many then binds the list of *all the received values* to x. Both receive and receive_many also work with send_reliable, and not just with send. The main difference is that send does not wait to be evaluated, while send_reliable waits until someone performs a receive or a receive_many. Thus, send is eagerly stepped:

$$\text{step}(f_1, \ldots, \text{send}[T](v); b) = f_1, \ldots, b$$

while send_reliable needs to wait:

$$\text{step}(f_1, \ldots, \text{send\_reliable}[T](v); b) = \begin{cases} f_1, \ldots, b & \text{when } \exists r_l \in e' : r_l = \text{receive}[T]() \\ f_1, \ldots, \text{send\_reliable}[T](v); b & \text{otherwise} \end{cases}$$

The semantics of sending and receiving are thus easily explained in terms of changes to the values (and the control flow) of other instances, with some added machinery to make sure that only the appropriate rules (`local`, `remote`, etc.) are run on each instance.

4.4 Reliability

One aspect that has not been covered so far is handling of transmission failures. All primitives but send_reliable never fail. send_reliable, on the other hand, may fail when the receiver is either disconnected or unreachable. After a certain time-out, send_reliable will return `false` to denote transmission failure, and inform the other instances that the receiver should be considered disconnected. This means that send_reliable are implicitly used for forcing disconnection of unreachable instances.

4.5 Asymmetry and load sharing

The ability to determine which instance of the game runs which entities locally allows us to take advantage of asymmetry for performance purposes. By estimating the network and CPU performance of an instance, we can determine its current overall performance. Overall performance of an instance could be used to have that instance create, and handle, more performance-intensive entities, such as those with complex AI, physics, etc. This way instances that have more processing power (for example because of better hardware) or bandwidth (because of a better local network infrastructure) would be used to lighten the load of the "weaker" instances.

This topic is quite broad and complex, and it is outside the scope of the current work. For this reason we do not expand it in depth, but leave it sketched.

4.6 Closing remarks

This concludes the presentation of Casanova syntax and semantics. The syntax may look, at a first glance, deceptively simple, but in reality it drives a complex translation. All networking operations are then translated away in a series of list operations on incoming or outgoing mailboxes. The various mailboxes then handle messages, which are responsible for the low-level sending and receiving of data through some simple, low-level, underlying networking library that is required to provide little more than sockets and connections.

5 Evaluation

We now discuss the main issues that we originally set out to address (see Section 2). Some issues are addressed through an analytical evaluation, some are based on rigorous performance benchmarks, and some are based on user studies.

5.1 Comparison with existing approaches

As a point of comparison, we have considered three representative networking samples: a tiny C# game with the Lidgren networking library, a chat program in Erlang, and a chat in C# with Windows Communication Foundation (WCF). Each sample provides a different target of comparison: *(i)* the C#/Lidgren sample is a low level framework that only supports basic connections and transmission of primitive data across the network; *(ii)* Erlang is a highly concurrent language centred around networked communications and concurrency; and *(iii)* WCF is an extremely high level system for handling remote communications.

Erlang is a concurrency and networking-oriented language. It excels at almost anything related to networking, besides two aspects: *(i)* dynamic typing is wasteful in terms of performance, severely limiting the usability of the language in game clients; and *(ii)* it does not have a concept of local/remote entities, requiring the developer to build it by hand for every game.

C#/Lidgren, being a very low level solution, barely simplifies any networking-related task. There is no support for concurrency (in terms of automatically dispatching messages to the appropriate listener), nor is there any support for fast automated serialization. Finally, there is no support for tracking locally and remote entities. To counterbalance these negative aspects though, Lidgren allows programmers to hand-optimize almost everything, to great potential in terms of performance.

C#/WCF is a very high level solution. This means that it tends to lack in performance, because of the use of reflection-based serialization, and it has no support for local/remote entities. Besides this, though, it is quite simple and immediate to use, if one ignores the (usually automatically generated) very large configuration files that it needs.

The results are summarized in Table 5.1. In conclusion, Casanova offers a set of primitives that are similarly powerful to those of Erlang or other high level solutions, with a performance profile that is more appropriate for games, and with additional syntactic and semantic support for the central game-related concept of local/remote entities.

5.2 Simplicity and expressive power

Simplicity and expressive power are two perspectives on the same issue. A good programming language should be *expressive enough* to express solutions in the chosen problem domain. A good programming language should also be *not too much expressive* as to offer too many subtly overlapping ways to solve the same problem.

Language	Concise	Serializes	Concurrent	Performance	Local/remote
Erlang	Very	Yes	Yes	Medium (dynamic typing, no incremental updates)	No
C#/Lidgren	No	No	No	High	No
C#/WCF	Partially (large config files)	Yes	Yes	Medium (slow serialization)	No

Table 1: Comparison with existing approaches

When a language is not expressive enough, then either some programs cannot be written in it at all (for example a language lacking recursion mechanisms or unbound loops), or (as is more often the case) some programs become very hard to build.

Casanova networking is both simple and has the right expressive power. Expressive power has been assessed by building a series of different games with networking in Casanova. The games belong to multiple genres: a shooting game, an RTS, an RPG, and a fighting game. Even if the games have been built at a simplified level (building full-fledged AAA games would not be feasible in a research context), they feature many important aspects of networking: lobby mechanisms, interpolation/extrapolation, limited use of bandwidth, fault tolerance and player disconnection, etc. The primitives offered by Casanova are amply sufficient to express all of these mechanisms with *relative ease*. Relative ease means that networking is still a complex domain, which requires reasoning in terms of many instances at the same time: the underlying thinking processes that the developer must perform remain quite complex. The games are available as Open Source Software at [**?**].

Simplicity has been assessed with a user study performed with a group of first, second, and third year game development students at NHTV University of Applied Sciences. All students were required to study and understand the behaviour of the samples above. After a given amount of time they received a questionnaire that they filled in. The questionnaire measured their understanding, which on average was high enough to be satisfactory with regards to the understandability of the language. Given the short time frame of the experiment, and the lack of previous knowledge of Casanova, we believe that in this case *understandability strongly correlates with simplicity*. The questionnaire is also stored online at [**?**]. The results of the test are found in Table 5.2.

1	2	3
4	5	6
7	8	9

Table 2: Questionnaire results

1	2	3
4	5	6
7	8	9

Table 3: Comparison with other libraries

We also observe that simplicity in Casanova networking comes from a specific factor. The number of primitives offered is really small. Four keywords for sending and receiving, and three keywords for specifying ownership and connections. A total of seven keywords is really a small set for something as big as building multi-player games. Existing libraries, when compared to Casanova networking, are far larger, at least an order of magnitude, when counting the number of primitives they offer. In this case, we counted non-trivial methods as primitives. The results of this comparison are summarized in Table 5.2.

5.3 Robustness

The ability of a game to function even in the presence of network issues or client disconnections is a very important issue. Wireless or mobile networks are unreliable, the Internet loses many messages that are sent with faster UDP connections, and game instances are often (purposefully or by accident) disconnected from the game.

Most of the networking primitives of Casanova simply never fail. Only `send_reliably`, on the other hand, blocks a rule execution until the networking operation is not completed. Casanova offers a rather aggressive, but reasonable, behaviour in the case of such failures (which have a large enough timeout to ensure no premature response!): the unreachable party will be forcibly disconnected from the game, and the other will have a chance to abort execution of the rule because at that point `send_reliably` will return `false`. Of course it is possible to add manual handling of these failures, in order to avoid flat-out disconnections, for example by writing:

```
{
  send_reliably<T>(x)
  ...
} || {
```

1	2	3
4	5	6
7	8	9

Table 4: CPU usage

```
  wait(timeout)
  ...
}
```

In this case we are requiring that all operations are completed within a reasonable time-out, or else we locally abort rule execution without forcing the other party to disconnect in case of failed transmission.

Finally, when a peer disconnects, then Casanova automatically removes it from the list of peers. This guarantees that we will not send or receive anything to and from this peer any more. Disconnection may result in *orphaned entities*, which are entities that are only stored as slaves in all available instances. The orphaned entities can be handled with a brief voting session, where the first player to claim the entity will assign it to himself and signal to the others that they must connect to the new master entity. Alternatively, the various instances may be programmed to simply discard the orphaned entities, for example the avatar of a disconnected player.

5.4 CPU usage

The overhead of the Casanova networking libraries is rather small. As a benchmark, we have run a dummy instance of a game where all networked primitives immediately return a predefined value, instead of performing any actual networking operation. The difference in time for a single tick of the simulation was, on average, quite small. The results are summarized in Table 5.4.

5.5 Bandwidth and latency limitations

Bandwidth usage is a difficult aspect to assess, as it is heavily dependent on implementation details. For example, an application that often relies on `send<T>` for entities `T` which contain large amounts of data is going to use more bandwidth than an application that does the same less often. In turn, an application that only sends primitive values such as `send<bool>`, `send<int>`, and `send<float>` and which does so seldom, will use even less bandwidth. In short, depending on the requirements of the application, for

1	2	3
4	5	6
7	8	9

Table 5: Bandwidth and latency

example running inside a fast LAN or across the Internet, developers may choose a heavier or more optimized approach.

We show that it is *possible* to reach very limited bandwidth usage. We built the samples [**?**] with limited bandwidth in mind. Usage is well within the suggested maximum average bandwidth required for modern games. Bandwidth usage also heavily influences latency. Few transmissions, uncongested, result in lower latency. The results are summarized in Table 5.5.

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.