Distributed Systems 2 - First Assignment

Riccardo Capraro - 203796
Riccardo Micheletto - 215033

November 30, 2020

1 Introduction

The purpose of this report is to present our implementation of the *Broadcast-Only Communication Model Based on Replicated Append-Only Logs* algorithm proposed by Christian F. Tschudin [1], and analysis of some of the protocol components we tested.

The algorithm is based on the concept of solitons, waves generated by a station (relay) that propagate indefinitely in all directions of space as perturbations. Each wave, in addition to carrying the information it is propagating, is also composed of an identifier of the relay that generated it and a reference that allows to establish an order between waves generated by the same source. All this information is necessary to guarantee the system the properties of a reliable, ordered broadcast system. The algorithm then suggests to keep track of the received waves through the use of append-only logs. This data structure is useful to obtain some additional functionalities that might be desirable, such as the recovery of lost information.

This report is structured as follows: section 2 of the document will present the architecture chosen for the implementation and will outline the main components of the project, which was developed in Java using the Repast Symphony tool. Section 3 explains how we guarantee the protocol correctness. In section 4, the aspects related to the execution of the simulation will be presented: the description of the parameters that the user can adjust and the displays and graphs we introduced. In section 5 an analysis of some particularly interesting aspects that emerged during the development of this project and particularly relevant to this specific implementation of the algorithm will be discussed. Finally, in section 6 our conclusions about the results obtained will be drawn, while section 7 provides a brief guide for the installation and execution of the project.

2 Solution design

The proposed implementation supports the creation of a number of nodes randomly arranged in space, which interact with each other through waves with

a certain propagation range. The system is able to handle the dynamic addition of nodes during execution, random latencies in message exchanges, and the random loss of waves. Note that the original formulation of the algorithm involves waves propagating indefinitely, which is a subcase of our solution; in fact, also the authors of the original work describe the protocol as based on local exchanges.

Particular attention has been given to the management of wave propagation times, to make the system as close to a real world wireless model as possible. The waves therefore propagate in space in times directly proportional to the covered distance, reaching first the nearest relays and then the farthest ones. In order to achieve greater realism, random delays are added to the propagation times, to simulate the behavior of actual transmission systems. These random delays can cause the delivery of waves in wrong order with respect to the generation order, but the implementation of the frontiers proposed by the algorithm allows their correct management and guarantee the required ordering.

Below are the details of the main components of the project.

2.1 Model Builder and Configuration

The **ModelBuilder** class is responsible for the system initialization. It takes care of creating a **ContinuousSpace** where to uniformly distribute the relays and create a network that is used for the graphical representations shown during the simulation.

The **ModelBuilder** is also responsible for checking that the propagation range of the waves is large enough to allow all the relays to be reached by the perturbations. In the case of some nodes being isolated, the wave propagation distance is opportunely increased.

Additionally, the **ModelBuilder** dynamically adds new nodes to the system during the execution. The number of nodes added depends on parameters provided by the user, which will be better explained in 4.1. These parameters, along with all other parameters, are managed by the **Configuration** class.

2.2 Wave

The **Wave** class represents a perturbation that propagates in space and carries information. In the proposed implementation of the algorithm, each wave stores the identifier of the last relay that amplified and forwarded it, as well as the tick in which the receiver will have to process it, according to the propagation delays.

The **ProtocolWave** class represents the waves that are normally generated and sent during execution. In particular, these waves also carry the identifier of the relay that generated them, the reference and a numerical value.

Waves can be lost during transmission. A relay can issue a **RetransmissionRequestWave** to its neighbours; the message carries the relay frontier and is checked against the neighbour log. At the reception of a **RetransmissionRequestWave**, the log is checked for lost messages, and the neighbour sends

to the issuer a **RetransmissionProtocolWave** per lost message. This class, however forwarded the same as a normal **ProtocolWave**, is used inside the issuer to distinguish it from the latter, since normal **ProtocolWave** requires slightly further processing that is unnecessary in case of a retransmission.

2.3 Relay

The **Relay** class is the main class and it represents the station involved in the transmission of the waves and, through a series of methods opportunely scheduled in the **schedule** method, it manages the generation, reception and forwarding of all perturbations, following the algorithm specifications.

When creating the relay, the **addNeighbour** and **computeNeighbourLatency** methods allow to maintain a list of the nodes reachable by the perturbations, with the respective latencies due to the wave propagation times.

The **generateWave** method generates a new wave, which transports to the nearby nodes the appropriate information required by the protocol and explained above. The **forward** method is the one that concretely takes care of delivering the wave to the nearby nodes through their **receive** method, after having opportunely set the tick in which the wave will have to be processed, in accordance with the propagation times and introduced random delays. The receive method will insert the received waves in a priority queue, so that they are processed at the correct time and in the correct order. The waves are processed by the **onSense** method and forwarded if necessary via the **forward** method, which, following the protocol specifications, will update the frontier and generate a new perturbation, to amplify the original one and allow its propagation.

The **outOfOrderWaves** data structure, together with a series of methods, allows to manage waves arriving late due to random delays, keeping in memory those whose references do not correspond to the value of the frontier.

The **log** data structure contains the history of the network as seen by the relay. We defer the logging of waves by one period of time, and store them in a temporary data structure called **lastSentWaves**; this allows us to increase the performance of the retransmission protocol, by not retransmitting waves that may have been sent but yet not received by our neighbours. Eventually the log will be consistent with the information contained in the relay frontier.

Finally, the **Relay** class also includes the methods that compose the retransmission mechanism, where each relay periodically communicates its frontier to its neighbours, receiving all more recent waves the neighbours are aware of.

2.4 Style

The **styles** package contains all the classes used for the graphical representations shown during the simulation. In particular, three main views are proposed: *Network*, where a graph with all the messages exchanged between the nodes can be seen and where the wave propagation generated by a particular relay is highlighted, and *Send Load* and *Receive Load*, where the nodes and the edges

of the graph assume different colors depending on their load, i.e. the number of waves sent/received from the nodes or propagating on the edge.

2.5 Analysis

The **analysis** package contains the classes used to create some data sources that are plotted during the simulation. The classes implement the **AggregateDataSource** class and allow to collect statistics on the number of waves generated and received, as well as to calculate the average and maximum latencies of all the waves that are propagated in the system. We point out that cleaner approaches may be used.

3 Protocol properties

Three are the properties that we are required to guarantee: (1) perturbations have a unique global identifier determining their original source, (2) perturbations eventually reach all observers, and (3) observers sense perturbations coming from a specific source in the same order. Below is the explanation of how we enforced each property in our proposed implementation.

Unique ID The identifier of a wave is given by the pair (sourceID, reference); the sourceID is a unique identifier for the Relay that generated the wave, whereas the reference is an incremental counter used to order waves coming from the same source.

Eventual global delivery Waves are delivered at a relay if the next expected reference for the wave sourceId contained in the relay frontier is the the same of the wave. Note that the log, i.e. the history as seen by the relay, despite being updated in delay wrt. the frontier, will eventually share the same information.

The introduction of latencies in waves transmission can result in waves being delivered out-of-order to the receiving relay: we solve this problem by storing for later processing the waves with higher reference than the one expected, i.e. as soon as the wave with reference r is delivered, all out-of-order waves with reference $r\prime < r$ and same source are also delivered; note that this is sufficient to guarantee eventual global delivery when no new relay joins the network and when no message loss is possible; if this is not the case, then the introduction of the retransmission protocol is necessary to guarantee eventual global delivery.

Waves can be lost during transmission: the retransmission mechanism that we introduce ensures that all relay who miss some perturbations are able to retrieve them from their neighbours; moreover, we exploit the retransmission mechanism when node join the network too, to retrieve the most updated history: this ensures that also joining nodes are eventually aware of the whole history. In particular, retransmission allows node to join the network even after waves have stopped propagating (e.g. because we do not generate waves anymore).

In-order delivery Waves are delivered in order per design. A wave can be delivered only if its reference is the next reference the relay expects for the wave sourceID. The retransmission protocol works by exchanging normal protocol waves, i.e. retransmission and normal waves are delivered under the same constraints.

4 Simulation

4.1 Parameters

We now present the list of parameters that a user can modify. By setting the **default random seed** one can consistently e.g. reevaluate the same run of the protocol. **Space height** and **Space width** can be used to change the space dimensions. Relays can join the simulation at every period, starting from the beginning of the simulation, with a probability **relayJoinProbability** that a uniformly distributed number in the range [0,**relayJoinMaxNumber**] joins, up until the **relayJoinMaxPeriod**. Wave generation is bound by the **wave generation distribution**: we allow the user to specify custom uniform, normal and exponential distributions. While message exchanges happen at every tick, other important operations take place only on period multiples (retransmission requests, nodes joining the network, wave generation, neighbours update). A a period of the simulation is defined as a number of ticks, computed as 1 over **tick accuracy**. When waves propagate, they can be lost during transmission (with probability **lossProbability**) and always experience a random delay in the interval [0, PERIOD * **latencyRatio**]. Finally, we introduce four different colors to represent link loads: yellow, orange, red and black. The color is computed as the target value (e.g. receive load) times the **loadColorScale**. Finally, the parameter **waveGenerationLastPeriod** allows to stop waves generation in order to check the algorithm correctness and convergence time.

4.2 Displays

Below is the explanation of the displays that are shown during the simulation. We point out that while many features where initially introduced, we chose to focus on the ones that can be used to better understand the protocol behaviour.

Network shows nodes and the exchange of messages in the network. In particular, black edges represent message exchanges, whereas we show in red the propagation of messages generated from source 0, to point out how a single propagation takes place. We also introduced colors to express node behaviours (generating, forwarding, idle), but we deem this information irrelevant for our conclusions and hence we omitted it in the chart description. For an example see Figure 6.

SendLoad shows the load in terms of messages received by nodes. The color, as explained in the parameters section, is a visual indicator of the node or link loads. It is not relevant to show the exact number of waves, since the user can simply double-click on a given link or relay to show the values at runtime. Links load implementation is shared with the ReceiveLoad display, and shows the number of waves that have been sent by nodes but not yet delivered at the destination. We thought that representing waves as circles was less informative, and that makes showing what nodes are reachable from a given relay more difficult. For an example see Figure 2.

ReceiveLoad The difference with the SendLoad display is that we color nodes based on the number of waves they receive at a given tick. As already mentioned, the visual difference is minimal, and we did not focus our attention on improving this feature since all conclusions can be easily drawn from our charts. For an example see Figure 3 and 4.

4.3 Charts

We introduced several charts using Repast Symphony built-in capabilities to test the system correctness and performance.

Generated/delivered waves This chart shows the global number of generated waves and the number of globally delivered waves. A wave is globally delivered at time t' if it is generated at time $t < t'$ and at time t' at least the relays present in the network at t have delivered it. To ensure that joining nodes contain the whole history too, we simply check that all relays frontiers/logs are equal at the end of the simulation. For an example see Figure 1.

Load In the load chart we plot several information. The average number of neighbours a relay has, i.e. the average number of relays a wave generated from a given node can reach. We then introduce several metrics: **average send load** and **average receive load** tell us how many messages are exchanged and received; note that the number of received waves is usually higher then the number of sent waves, since a sent wave results in all neighbours receiving it. The **waves with delayed delivery** represents the average number of waves that have been received yet not processed because another reference was expected; this number increases over time if messages are delayed or lost, and is directly controlled by the the retransmission protocol and by the average number of neighbours (explained below). For an example see Figure 3.

Latency the latency chart shows the computed delivery latencies, defined as the time it takes for a wave to be delivered to all relays. The chart is updated as soon as waves are globally delivered, showing both the average delivery time and the maximum delivery time. For an example see Figure 2.

5 Analysis

5.1 Correlation between parameters and latency

The first aspect that our analysis wants to highlight is the correlation between some implementation choices and the delivery latency, defined as the time it takes for a wave to be delivered globally. In particular, latencies have been computed in number of ticks, and based on the fact the assumption that a perturbation travels exactly one *wave propagation distance* in one period, the period being computed as 1 over the *tick accuracy*. These two parameters highly influence the protocol's performances: when the tick accuracy is low, e.g. 0.1, the number of quanta of time when the waves can be received are low (10 per period in this case); this results in many waves reaching the relay at the same time: propagation latency therefore becomes less relevant, and the number of out-of-order waves will be decreased; however, since more waves are received concurrently, the *bandwidth* of the relay, namely the maximum number of waves it can send per tick, has a higher impact on the overall delivery latency of the protocol. Conversely, if the tick accuracy is high, e.g. 0.01 (100 ticks per period), even small propagation latencies can result in many waves being delivered out of order; accordingly, the limit set on the relay *bandwidth* is less relevant, since waves deliveries are spread across a bigger number of ticks. An example of this phenomenon is shown in figure 2.

5.2 Wave generation probability distributions

As explained in 4.1, the user has the possibility to change the probability distribution used by the system for the wave generation. From figure 3 it is possible to see how different distributions have an actual impact on waves generation, and in particular on sent and received waves. Apart from the probability distribution, the three simulations share the same parameters and represent a basic scenario where no relays dynamically join the system.

5.3 Protocol robustness

The peculiarity of this protocol is its robustness. Uninformed broadcasts are indeed inefficient: a huge amount of unnecessary information is exchanged between relays, as they are not aware of which of them already received each message. However, this redundancy bring robustness to the algorithm, as some wave losses are resolved by the reception of duplicate information.

Robustness can be appreciated in the scenario where the wave propagation distance assume two different values, leading to different amount of redundant information sent. With a larger propagation distance, each relay can reach more neighbours and produce a greater amount of waves. As can be seen in figure 4, the number of retransmissions is lower when the propagation distance is higher, as in the latter case the number of (redundant) sent waves is greater.

Redundant waves also allow to solve more quickly the reception of out of order waves. Because of random latencies, the reception of waves with lower reference can be delayed, resulting in the ordering required by the protocol not being respected. For this reason, out-of-order waves are temporarily stored in a data structure, and processed only after the reception of in-order waves. Redundant waves can provide missing in-order waves and allow a faster delivery. Figure 4 also shows how a bigger number of waves exchanged allows to keep a lower number of out-of-order waves.

5.4 Propagation latency

The propagation latency, i.e. the time required for a wave to reach all the relays, is closely related to two factors. The first one is the maximum number of messages that a relay can send every tick or, in other words, the available bandwidth of the relays. With a low bandwidth some wave amplifications could be delayed, impacting on the algorithm performances and on the time required to reach all the nodes. In figure 5 it is possible to notice how a lower bandwidth cause an increase on the achieved global delivery latency.

The second factor that influences the performance of the algorithm is the topology of the system. The way the relays are arranged in space in fact affects the path of the propagations: therefore, the maximum propagation latency is directly correlated with the diameter of the graph induced by the wave propagation distance. With the same parameters, ad in particular with the same relay density and the same space surface, in figure 6, where the diameter of the network is higher, it can be seen that the latency is also higher. On the other end, low diameter topologies, such the one in figure 7, lead to better latencies.

6 Conclusion

After several experiments we can conclude that the protocol could be used in a real implementation instead of a normal wired network. In fact, since the relays do not need to know about the existence of neighbours nor their location, but simply need to broadcast signals, the setup of the network would require little configuration. The only parameters that should change based on the network topology and the relays physical distance is the wave propagation distance: indeed the larger the number of neighbours, i.e. relays that can be reached by the propagation, the more robust and faster the protocol is. An improvement on our implementation is to modify the wave propagation distance dynamically, based on the number of neighbours: it can be shown that nodes that are part of the core of a network require smaller wave propagation distances to be reached then nodes at the edge of the network. A mixed network with a few more powerful relays (with better antennas and higher wave propagation distance) can be enough to guarantee low latencies even with very specific topologies.

7 How to install

To run the code you can either import the project in eclipse or run **java -jar installer.jar**, that can be downloaded from `https://drive.google.com/file/d/12TQqRyzfIqjsmM1PgoXr1_2jKD0pTYzI/view?usp=sharing`. Since the project was compiled with java11, be sure to use a compatible version. When you run the command above, an installer will allow you to install the program on your system: choose a folder and there you will find the program *start_model* that you can use to start the model.

References

[1] Christian Tschudin. A broadcast-only communication model based on replicated append-only logs. *ACM SIGCOMM Computer Communication Review*, 49:37–43, 05 2019.

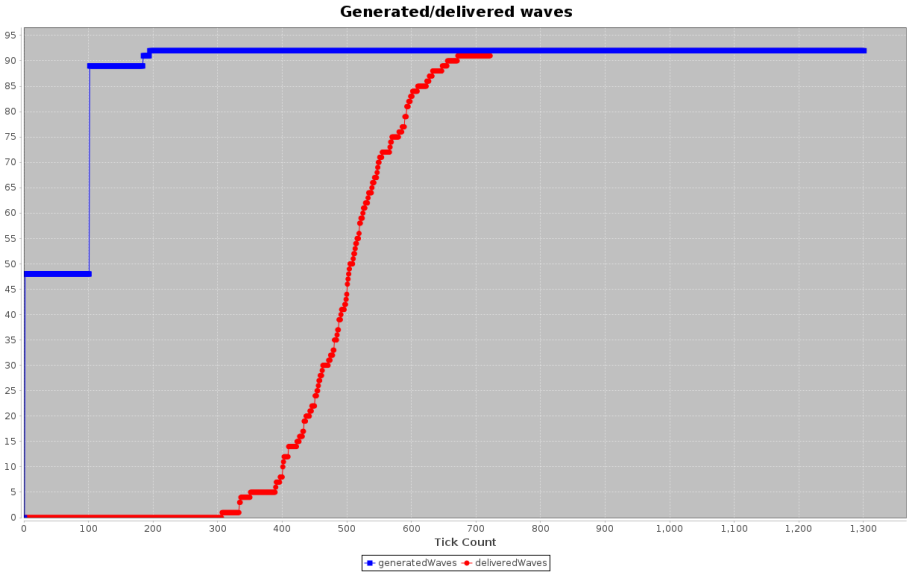
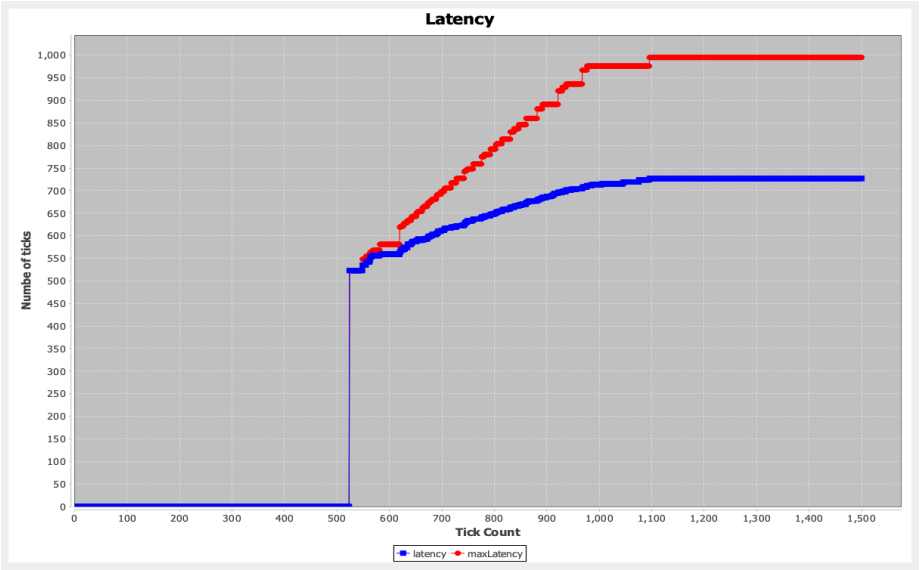
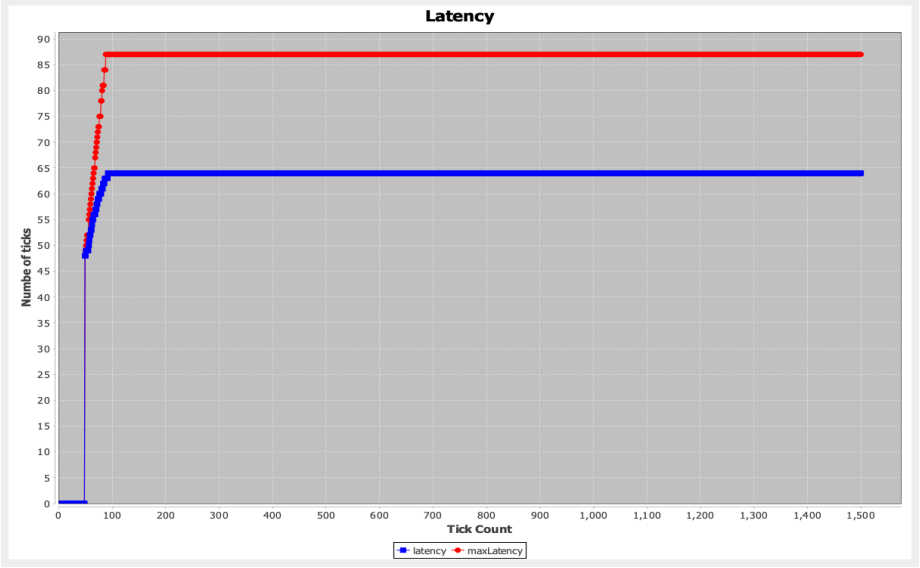


Figure 1: Globally generated and delivered waves chart

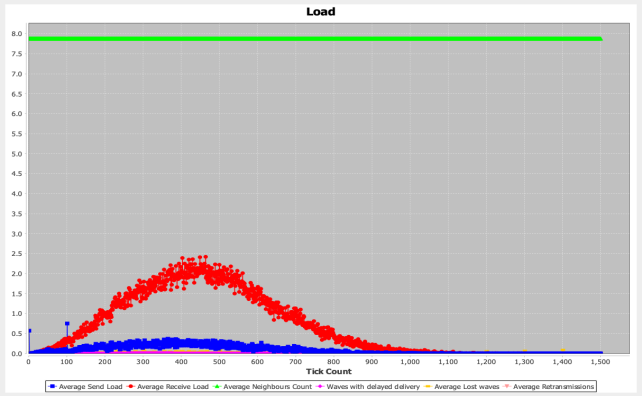


(a) Latency with 0.01 tick accuracy

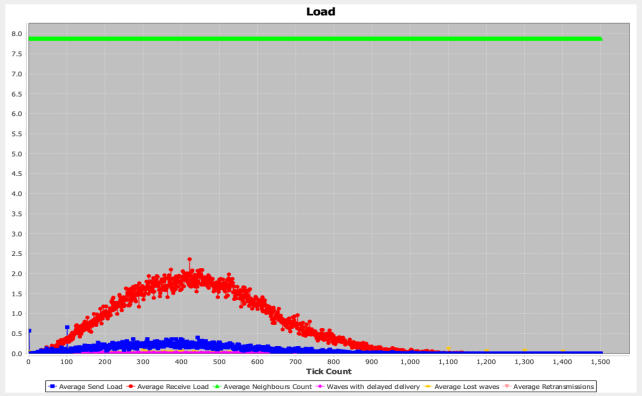


(b) Latency with 0.1 tick accuracy

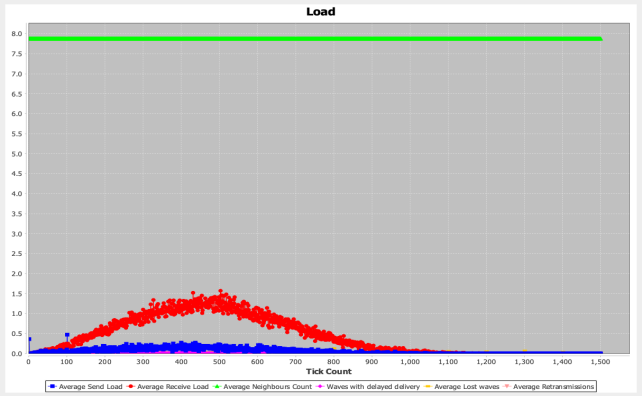
Figure 2: Latencies with different tick accuracies



(a) Load with uniform distribution in [0,1]

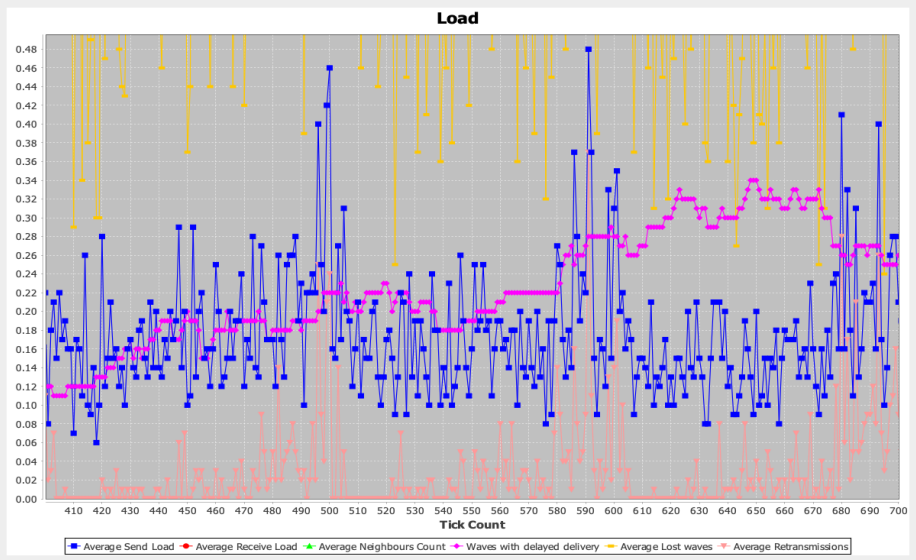


(b) Load with normal distribution, mean=0.5, variance=1

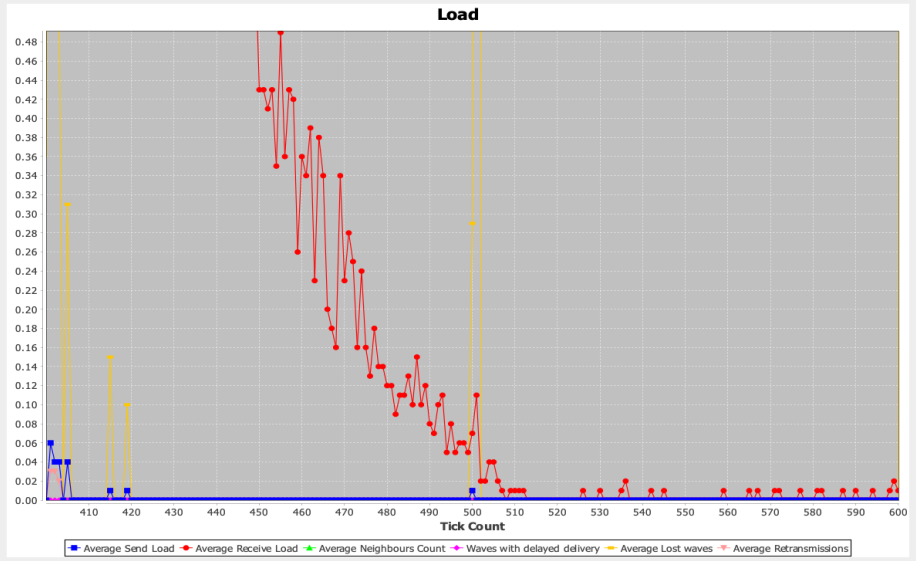


(c) Load with exponential distribution, lambda=2

Figure 3: Loads with different probability distributions

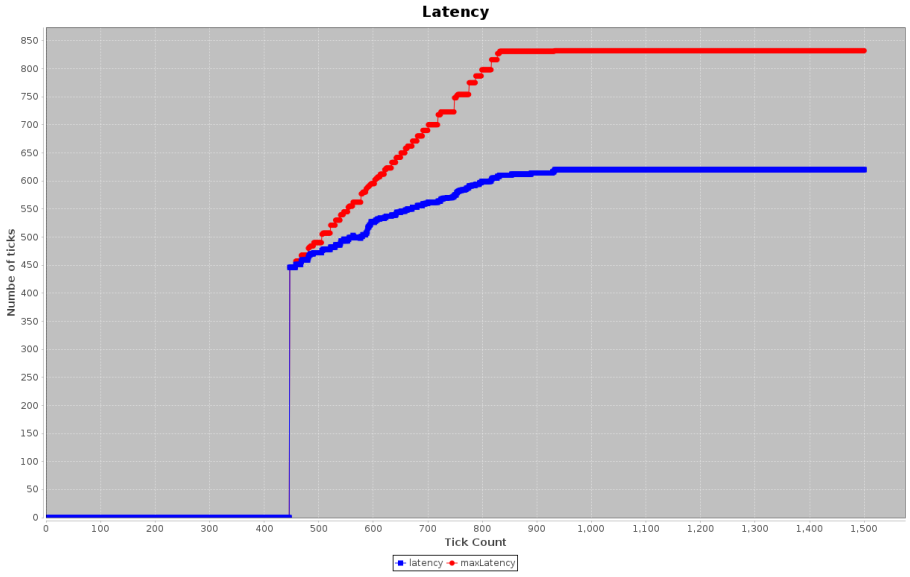


(a) Load with wave propagation distance equal to 8

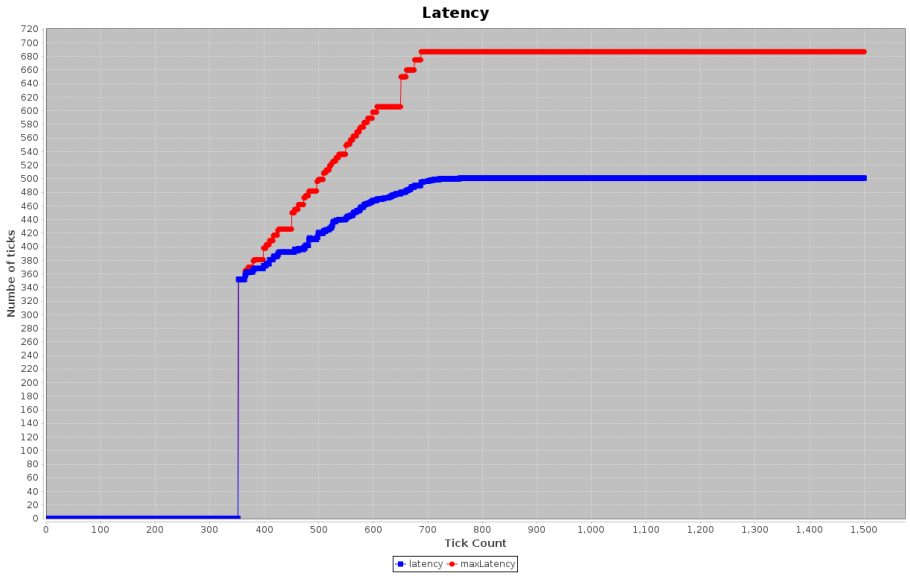


(b) Load with wave propagation distance equal to 20

Figure 4: Different retransmissions and out-of-order deliveries with different propagation distances

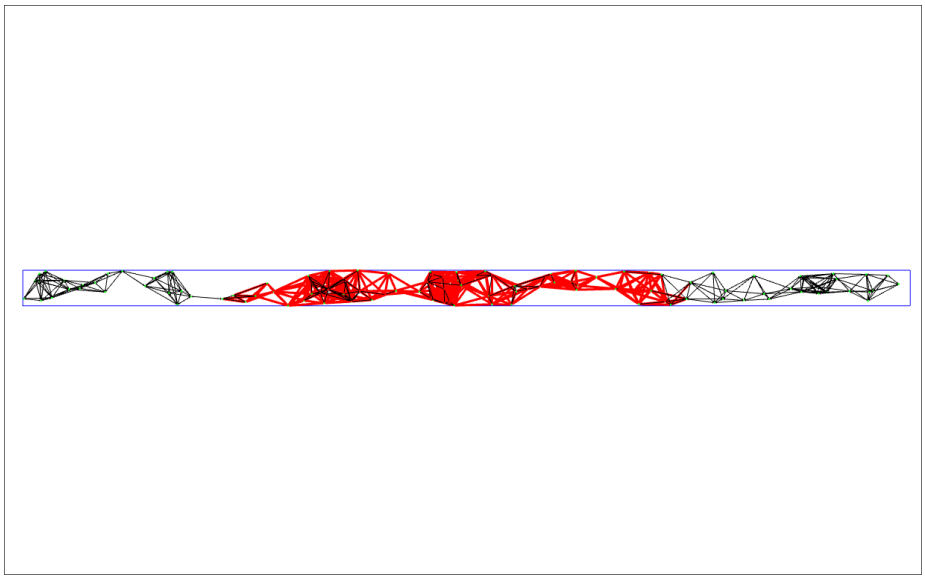


(a) Latency with maximum bandwidth equal to 10

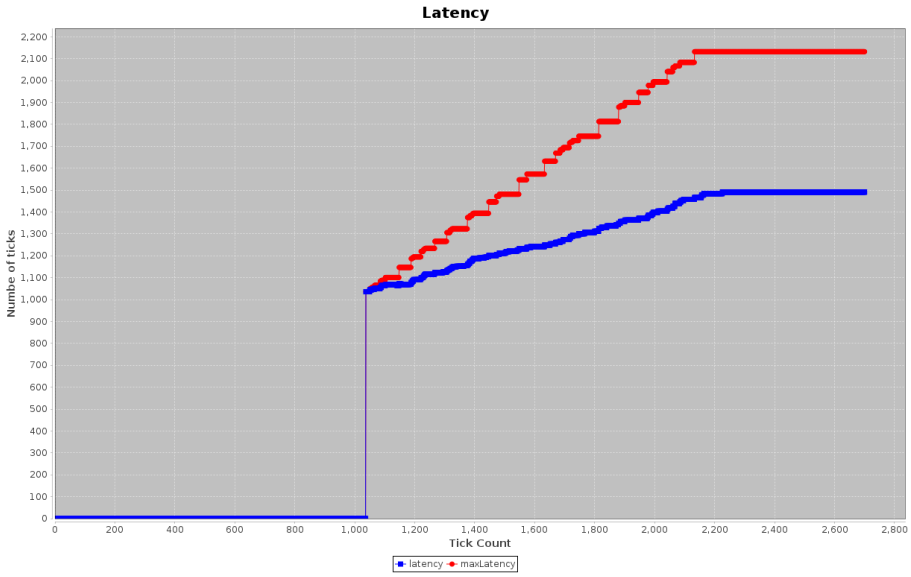


(b) Latency with maximum bandwidth equal to 500

Figure 5: Different latencies caused by different bandwidths

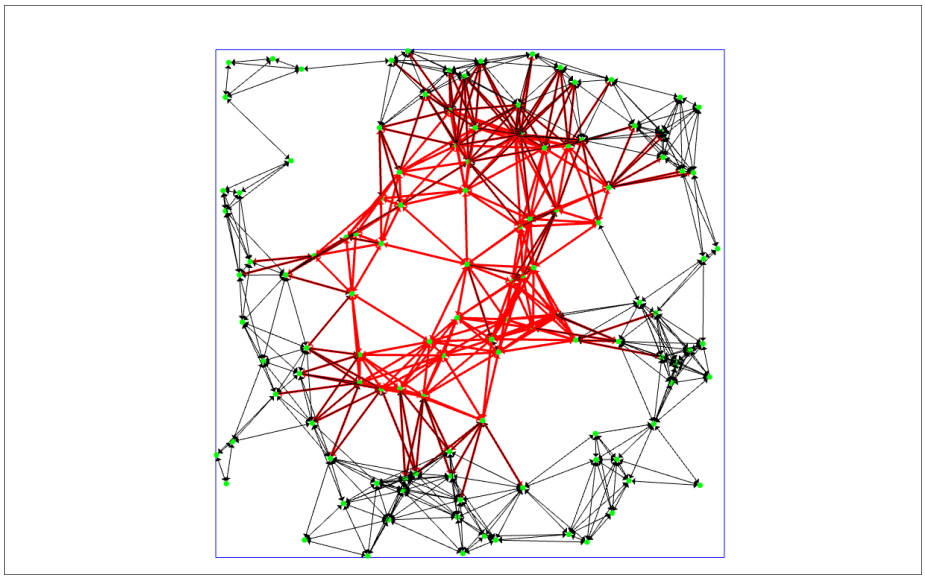


(a) Topology

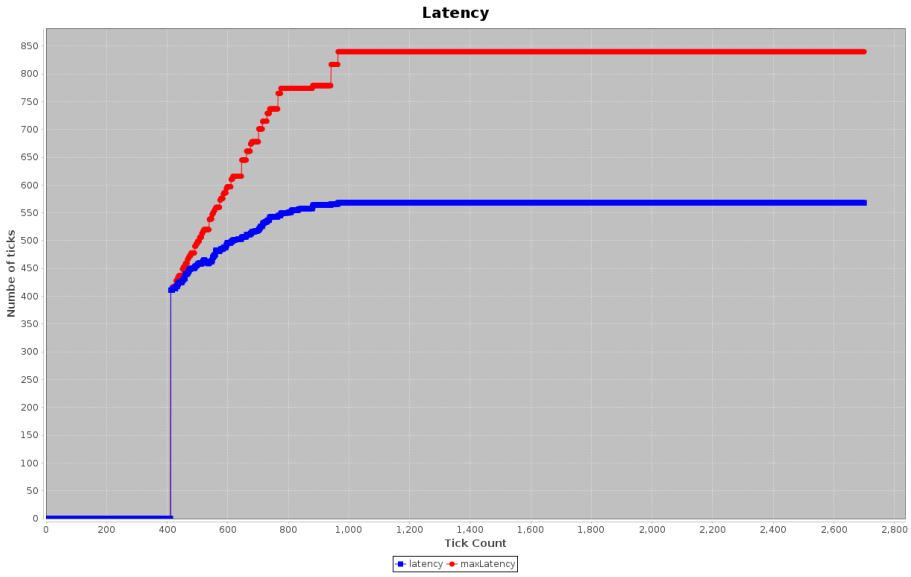


(b) Latency

Figure 6: Latency with high diameter network topology



(a) Topology



(b) Latency

Figure 7: Latency with low diameter network topology