Distributed Peer-to-Peer Simulation

Vasco de Carvalho Fernandes
vasaco.fernandes@ist.utl.pt

Instituto Superior Técnico
Av. Prof. Dr. Aníbal Cavaco Silva
Lisbon, Portugal

Abstract. Peer-to-peer overlays and applications are very important in current day-to-day applications. In the future this seems to be even more relevant. Peer-to-peer technologies bring benefits regarding decentralized control, resource optimization and resilience.
Simulation is an indispensable tool to help create peer-to-peer application protocols. Unfortunately current peer-to-peer simulators are flawed and unable to serve their purpose beyond the limitations in memory and performance of a single machine.
We proposed DIPS, a distributed implementation of the Peersim simulator to overcome these limitations. By utilizing technologies from parallel systems simulation, distributed agent simulation and peer-to-peer overlays themselves we aim at not only overcoming those limitations, but also increase simulation performance and scalability.

Keywords: peer-to-peer, p2p, simulation, Peersim, DIPS, distributed-simulation

1 Introduction

Peer-to-peer overlays and applications have had historical importance in the development of current network aware applications. In the future, the number of network connected devices is expected to grow exponentially, making peer-to-peer applications ever-more relevant. We will show the state of the art of peer-to-peer simulation, point out its shortcomings and propose a distributed peer-to-peer simulator, DIPS, to help developers overcome the challenges in creating peer-to-peer applications and protocols.

Peer-to-peer systems are distributed computer systems where network communication is done directly between endpoints, not requiring a central server as an intermediary. They oppose common client-server architecture that composes the large majority of network communicating systems today.

When a developer creates a peer-to-peer protocol, even if analytically deemed as correct, efficient and scalable, a test environment must be setup to evaluate the protocol's characteristics. Peer-to-peer protocols are usually designed to connect a very large number of nodes. In order to convincingly test the protocol, a simulation environment is necessary. Furthermore, a reduced scale deployment on real network settings is also required.

Current peer-to-peer simulation suffers from a peer-count limitation due to memory limits. When running a simulation on a single computer this limitation cannot be overcome. Other approaches, such as a virtualized deployment environment, have proven to be inefficient and unable to surpass the memory limit using reasonable amounts of resources. Hence the need for a custom-made solution aware of peer-to-peer simulation implementation characteristics. Only such solution could surpass the memory limit and still execute the simulation with acceptable performance.

Simulation environments such as Peersim [1] have a limit of nodes they can simulate. This limit could be overcome with a distributed Peersim. A distributed Peersim would allow the developer to simulate more nodes in his simulation.

2 Related Work

Peer-to-peer P2P systems are characterized by decentralized control, large scale and great dynamism of their population and operating environment. Decentralized control is the key aspect of a P2P system. Although variations on the level of decentralization exist between different system types, control over interaction between peers is never centralized.

The more relevant examples of peer-to-peer architectures include Chord [2], Pastry [3], CAN [4], Tapestry [5] e Kadmelia [6]. On top of this ones some applications were created [7,8], publish-subscribe [9,10]. Outside academia, Napster, Gnuitella, Kazaa, e BitTorrent are known examples of peer-to-peer application for file-sharing.

Applications base themselves on a peer-to-peer architecture primarily due to its capacity to cope with

churn and network failure. Such architectures are usually characterized by their scalability, no single point of failure and large amount of resources. True decentralized peer-to-peer systems do not have an owner or responsible entity, responsibility is instead shared by all peers. Peer-to-peer architectures also have the potential to improve and accelerate transactions through their low deployment cost and high resilience.

2.1 Simulation

Simulation is an important tool to test protocols, applications and systems in general. Simulation can be used to provide empirical data about a system, simplify design and improve productivity, reliability, avoiding deployment costs. Simulation testbeds offer different semantics and abstraction levels in their configuration and execution according to the level of abstraction desirable for each type of simulation.

Discrete Event Simulation A discrete-event simulation is typically a loop where the simulator will fetch one event from a queue, execute one step of the simulation, possibly update the queue and restart. Simulation is slower than the simulated systems.

Discrete-event system simulations are by their very nature sequential. Unfortunately this means existing simulations cannot easily be partitioned for concurrent execution as sophisticated synchronization techniques would be required to ensure cause-effect relationships.

2.2 Peer-to-peer Network Simulation

Peer-to-peer simulation is an abstraction from general network simulation. Simulating peer-to-peer protocols involves the transfer of messages between peers and the collection of statistics relevant to the simulation.

Current peer-to-peer simulators typically run an event-driven mode, which is a discrete-event simulation closely related to more general network simulation and to process simulation. Messages are sent between simulated peers, they are saved in a queue and processed in order by the simulation engine.

Another type of simulation is a cycle-based simulation. In cycle-based simulation each simulated component (the peer) is run once per cycle, whether or not it has work to be done. This offers a more simplistic simulation environment.

Peersim Peersim [1] is a peer-to-peer simulator written in Java. It is released under the GPL, which makes it very attractive for research.

Peersim offers both cycle-based and event-driven engines. It is the only peer-to-peer simulator discussed here that offers support for the cycle-based mode. Peersim authors claim the simulation may reach 10^6 nodes in this mode.

P2PSim P2PSim [11] is a peer-to-peer simulator that focus on the underlying network simulation. It is written in C++ and like in Peersim, developers may extend the simulator classes to implement peer-to-peer protocols.

The network simulation stack makes scalability a problem in P2PSim. P2PSim developers have been able to test the simulator with up to 3,000 nodes.

The C++ documentation is poor but existent. Event scripts can be used to control the simulation. A minimal statistics gathering mechanism exists built in to the simulator.

Overlay Weaver Overlay Weaver [12] is a toolkit for the development and testing of peer-to-peer protocols. It uses a discrete-event engine or TCP/UDP for real network testing.

Distributed simulation appears to be possible but it is not adequately documented. Scalability wise the documentation claims the simulator may handle up to 4,000 nodes, the number of nodes is limited by the operating systems thread limit.

The documentation is appropriate and the API is simple and intuitive. Overlay Weaver does not model the underlying network.

PlanetSim PlanetSim [13] is also a discrete-event simulator written in Java. It uses the Common API given in [14].

The simulator can scale up to 100,000 nodes. The API and the design have been extensively documented. The support for the simulation of the underlying network is limited, however it is possible to use BRITE [15] information for this purpose.

2.3 Parallel simulation

Parallelization requires the partition of the simulation into components to be run concurrently. Simulation of systems embodies this concept directly.

We can model a system as:

- System – A collection of autonomous entities interacting over time.
- Process – An autonomous entity.

– System state – A set of variables describing the system state.
– Event – An instantaneous occurrence that might change the state of the system.

Processes are the autonomous components to be run in parallel. However, the separation of the simulation into multiple components requires concurrent access to the system state which poses problems of synchronization.

Parallel discrete-event simulation of systems In parallel simulation of physical systems, consisting of one or more autonomous processes, interacting with each other through messages, the synchronization problem arises. The system state is represented through the messages transfered between processes, these messages are only available to the interacting processes creating a global desynchronization.

As discrete-event simulation is typically a loop where the simulator will fetch one event from a queue, execute one step of the simulation, possibly update the queue and restart. Simulation becomes slower than the simulated systems.

In systems where process behavior is uniquely defined by the systems events, the maximum ideal parallelization can be calculated as the ratio of the total time require to process all events, to the length of the critical path through the execution of the simulation.

2.4 Distributed Simulation

Distributed simulation differs from parallel simulation on a small number of aspects.

Distributed systems must take into account network issues related to their distributed nature, notably: Latency, Bandwidth, Synchronization.

Simulation of peer-to-peer systems is traditionally done in a sequential manner, and with the exception of Oversim no simulator offers the possibility of distributed execution, and this is more a foreseen possibility than an actual implementation [16].

Distributed simulation of agent-based systems Agent simulation is an area where distributed simulation environments are used extensively.

Agent based systems deployment areas include telecommunications, business process modeling, computer games, control of mobile robots and military simulations [17]. An agent can be viewed as a self contained thread of control able to communicate with its environment and other agents through message passing.

Multi agent systems are usually complex and hard to formally verify [17]. As a result, design and implementation remain extremely experimental. However, no testbed is appropriate for all agents and environments [18].

The resources required by simulation overcome the capabilities of a single computer, given the amount of information each agent must keep track of. As with any simulation of communicating components, agent based systems have a high degree of parallelism, and as with other particular types of simulation distributing agents over a network of parallel communicating processes have been proven to yield poor results [19].

3 DIPS Architecture

We propose DIPS, a Distributed Implementation of the Peersim Simulator. DIPS is a set of Peersim regular instances that run one global simulation where the simulated peers have access to each other.

In order for Peersim instances to be able to share one simulation that spans all of them, we must also provide the foundations of communication between instances so that simulation components have access to each other, and are able communicate with reasonable performance. We must take that concepts that are the basis of Peersim, extend them so that they can adequately be used in a distributed context. Finally we must guarantee that losses in performance of the simulation, due to the new distributed characteristics are minimized, and that challenges created by the distributed behavior, are met in a way that does not overburden the simulation creator.

The architecture of DIPS divided into three large aspects, one for each explained above.

3.1 Network

Network communication is a crucial factor for the performance and viability of DIPS. In a centralized simulator, communication is not a problem, as the whole simulation is run by a single process. As soon as more than one machine is introduced, network communication becomes inevitable.

Our approach in DIPS was to define a independent component of the simulator to encapsulate all network communication.

In a distributed simulator one of the most important factors of its design is to minimize the negative impact on performance that the overhead of network communication might produce. In the particular case of DIPS, as it extends the Peersim simulator, favorable comparisons can only arise if the impact of network delay can be compensated.

We defined the network communication component as an actor according to the actor model. This is not only a semantical isolation. Communication is executed through message passing, therefore communication structures are clearly defined. It is also a physical isolation, by removing shared memory from the design we guarantee independence of the simulator from the network, limiting the impact that network communication processing can have on the performance of the simulator.

The network component is the single point of external communication in DIPS. Every component that requires communication with other instances must go through the network component. The fundamental role of the network component can be described by the following actions:

- Given a message and an address, the network component guarantees delivery of the message at that address.
- Subscribers receive messages destined to their address on message arrival.
- If a message is received to an address with no subscriber, the network component will hold the message until the address owner collects it.

There are a few guidelines regarding the design of the DIPS network: 1) The organization of the network should be simple; 2) Communication should be efficient; 3) Virtual address lookup must be $O(1)$; 4)Organization should be versatile enough to handle a large number of instances if necessary.

The small number of instances involved in the process permits an approach where every instance knows of all others. This, *well know* behavior guarantees simplicity, and allows messages to be broadcasted to the entire network. As long as the number of broadcasted messages is kept to a minimum, and only used when strictly necessary, the performance should not suffer too much from this approach.

3.2 DIPS Event Based Simulator

The distributed event simulator is the core engine of DIPS. The distributed event simulator shares much of the functionality with its centralized counterpart implemented in Peersim.

A simulation is composed of two main parts, the initialization and the event processing loop.

The initialization phase creates the network. The distributed network differs from the traditional Peersim network in that it must differentiate from local and remote nodes, and disallow local access to remote nodes. The distributed network extends the Peersim network and therefore offers all services available in the Peersim simulator, most importantly random access to network nodes.

The provided random access creates a problem. Because local access to remote nodes is not available it would be required that all structures accessing the network know in advance if a node is remote or not. We believe this would cause an unnecessary burden on the simulation implementer, as well as create a an incompatibility layer with code written to the Peersim simulator, that could be avoided. The distributed network is design to offer a transparent layer to all components that is unaware of the distributed characteristic of the simulation, these components view local nodes as the complete network and access them through original Peersim APIs. Components aware of the distributed characteristic of the simulation have access to methods that present the entire network.

Code design for Peersim and naive implementations may iterate over local nodes, while more complex, network aware components can view the entire dimension of the network and differentiate between local and remote nodes.

The next step of the initialization is to run network initialization components. These components are executed exactly as in Peersim, their purpose is to initialize values and create links. Initialization components are particular cases of control components, just like these, they only have access to the local network. Access to nodes outside the local network is not possible, however being specialized controls, initialization components have access to the Control Communication API.

In Peersim control components are executed through scheduling, the configuration defines when a control should be executed in relation to the global time of the simulation. In DIPS there is no global time, each instance maintains a local time and there is no effort to synchronize clocks, instances added in the middle of a simulation will have completely incoherent clocks when compared to the other instances. The lack of global time was a decisive factor for the design decision to have control components be local instead of global. Each control component will run according to the defined scheduled in relation to the local clock, this component can only access the state of the local nodes.

Control components are cannot effectively control the simulation if they are only able to affect the local state of the simulation, for this reason we introduce Distributed Controls that have access to the Control Communication API. Distributed Controls may register a name in order to receive messages, whenever a message is received, the destination control is executed. We have defined a new execution method for

control through message passing, in addition to the scheduling.

The message processing loop serves as a centralized event loop that guarantees sequential processing of the simulation events. When considering simulation events we think further than the messages passed between nodes of the simulation. Simulation events processed by the Event Processing Loop include node to node messages, control to control messages through the Control Communication API and scheduled control executions.

Control events, both scheduled and message triggered, take precedence over the node-message processing and therefore are processed at the beginning of each stage of the Event Processing Loop, the execution of control events is discussed in the Control (see Control) and Distributed Control (see Distributed Control) sections.

Each stage of the message simulation loop corresponds to one clock tick, in each stage the simulator performs three tasks. First all control messages that have arrived since the last tick are processed this involves calling the *messageReceived* method of the control with the message, the control is then able to manipulate the simulation at will, e.g. create a checkpoint of the current simulation stage. After all control-messages have been processed the second task is to call the *execute* method of all control objects scheduled to be run at this tick. The third and last task of the loop iteration is to process the first message in the queue, this is done by calling the *execute* command of the message destination protocol of the message destination node.

4 Prototype

5 Evaluation

First we compare the DIPS prototype convergence to Peersim convergence. The intuition behind this test is, if the DIPS prototype produces the same results as Peersim, then the DIPS prototype can be deemed adequate as a peer-to-peer simulator.

We plotted the variance of an Average simulation of 10000 nodes where the nodes were initialized we linear values from 0 to 100.

This is an anecdotal example that does not proof the adequacy of the DIPS simulator, but does open the possibility for it to be inferred. If one accepts that the DIPS simulator running on one instance, is architecturally equivalent to the Peersim simulator than one can accept that, apart from implementation faults, simulations that converge in Peersim also converge in the DIPS simulator running on one instance.

If simulations converge on DIPS running on one instance, the obvious is what does it take for them to converge when run on more than one instance. By analyzing the architecture of DIPS it is possible to see that the message queue on each instance is key whether or not a simulation will convert. If remote messages (messages originating in a different instance) could be delivered locally instantly then the simulation would be equivalent to a centralized one which we have already accepted that converges.

It is the measure of how much delay remote messages have over local messages that define how well will a simulation converge on a distributed simulation. We will take measurements of this metric in the next section.

5.1 Local Clusters

We must test the DIPS prototype in relation to the creation of local clusters. Intuitively, because messages to local instance nodes generated locally, can be processed immediately, while messages destined to a remote instance must transverse the network, it is possible that local clusters occur.

A local cluster is a set of nodes, in the same instance that can send messages to each other with a smaller delay than to other nodes, therefore at a much faster pace. If these nodes are created, in a simulation that does not have a bias against local or remote nodes, we should be able to see a rise in the number of locally generated, processed messages in relation with the usually expected amount.

The Infection simulation is particularly good for this test as it generates message to local and remote nodes in a predictable way. The Infection simulation generates N messages per message received, where N is the configured infection degree. The number of messages generated for local and remote nodes is also known. As the nodes neighbors are linearly distributed from the simulated network population and messages are sent to a neighbor chosen at random, the expected number of messages sent to local nodes is equal to the proportion of nodes in the simulated network, i.e $\frac{1}{N}$.

For the this test an Infection simulation was run in a AWS medium instance with a combination of the following variables:

- Instance Count: 1, 2, 3, 4
- Network Size: 40000, 80000, 160000, 250000, 1000000, 1500000, 2000000
- Infection degree: 1, 3, 5, 8

The simulation was allowed to execute 50000 events in each instance, after which it was stopped and the

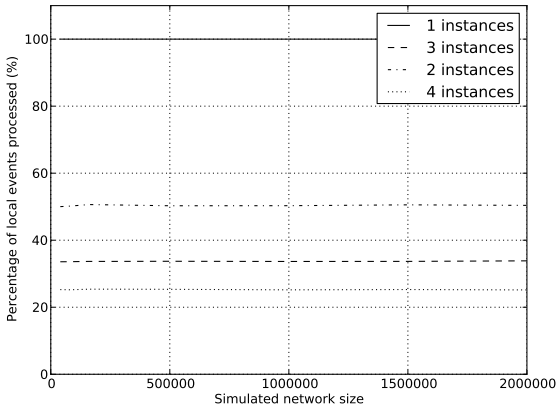

Fig. 1. Infection: percentage of local events with degree=1

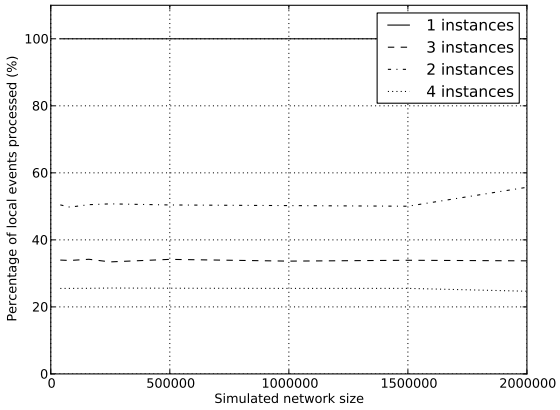

Fig. 2. Infection: percentage of local events with degree=3

number of local and remote messages processed in each instance was saved. The total number of local and remote events processed in the entire simulation was then calculated. We can see plotting results in figures 1 and 2 . Figure 1 shows the number of local events (to each instance) processed in the entire simulation per network size for a simulation with degree 1.

The results are the best possible outcome, the number of local events processed at each instance is equal to the expected value, in all configurations, a fraction of the total ($\frac{1}{2}$,$\frac{1}{3}$,$\frac{1}{4}$).

In figure 2 the same information is plotted but for a simulation with a degree of 3. This not only confirms the previous results, it also shows the we were not in face of a problem of starvation, even when a much larger number of messages is generated than the ones processed, we still have a correct balance between local and remote messages.

This test shows that the remote messages generated will eventually be processed, it says nothing of when the remote messages will be processed. In the next section we will test whether or not remote messages are processed with acceptable delay in relation to local messages.

5.2 Message Latency

The last test performed shows the fitness of the DIPS simulation, tests local and remote average message latency. In a distributed simulation some messages will be destined to nodes held in the same instance, which we call local messages, while others are destined to nodes held at remote instances, called remote messages.

The test setup is similar to the one described in the previous section. The Infection simulation was run on a varying number of instances, on AWS EC2 small instances. Several configurations were tested, in a combination of different message bundle sizes, simulated network size and infection degree. All tests were run at least three times, and the values averaged. The simulation was allowed to run for 50000 events on each instance, after which it was stopped.

Whenever a message was created, the creation time was timestamped. The initial timestamp was then used to calculate the message delay when the message was dequeued to be processed. Because remote messages were timestamped in a different instance from were they were processed we use a clock synchronization, to be able to compare average message delay between remote and local messages. Testing on the same hardware indicates the protocol error is close to 100ms. Each message delay calculated also an up-

dated average message delay for the instance. Separate average message delay were calculated for local and remote messages and were saved at 25000 events intervals.

The test configuration was created as a combination of the following parameters:

- Instance Count: 2, 3, 4, 8
- Network Size: 40000, 80000, 160000, 250000, 1000000, 1500000, 2000000
- Message Bundle size: 1, 100, 1000, 10000
- Infection degree: 1, 3, 5, 8

Ideal results would show no change between local and remote average delay in any simulation configuration. More realistic expectations would be to have some configuration type where the difference between local and remote messages is negligible so that this information can be used in the future to tune simulations.

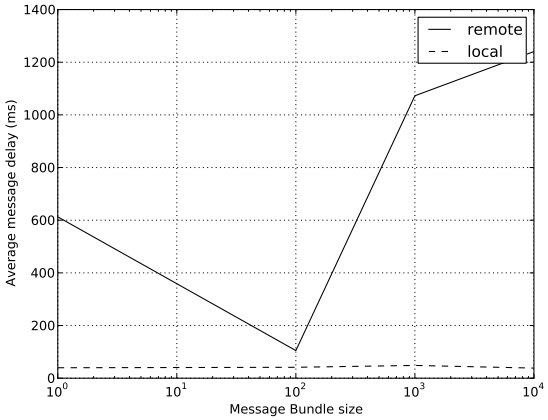

Fig. 3. Infection: comparison of local and remote average message delay in simulation with degree of 1

Results show that some of the variables in the simulation are orthogonal to the average message delay. Instance count and network size do make a difference in the average message delay. Infection degree and message bundle size, however are responsible for greater differences in the message delay.

In Figure 3 we plotted the average message delay in function of the message bundle size for a simulation with a degree of 1. The graph indicates serious issues in network throughput for a message bundle size either to small or too large, in later sections we will confirm this problem. The most interesting result here is that there is a message bundle size where differences between local and remote average message delay are almost negligible, under the clock error.

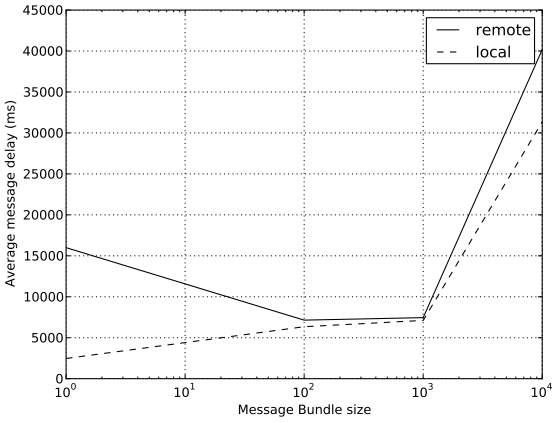

Fig. 4. Infection: comparison of local and remote average message delay in simulation with degree of 3

In figure 4 the same variables are plotted but, this time, for a simulation with degree 3. This graph confirms the previous one, and data from simulations with different degree behave similarly. Unfortunately we can see a pattern that makes it harder to define which are the perfect conditions to run the simulation. While on the simulation with degree 1 only a message bundle with size equal to 100 gave good results, here both 100 and 1000 are good candidates, with 1000 doing fairly better. Data indicates that the best message bundle size depends on the degree of the simulation, *i.e.* it depends on the number of messages in the queue.

5.3 Memory Usage

The last objective of DIPS is to overcome the memory limitations inherit to centralized simulators. In this section we will test the memory usage of the DIPS prototype, both in relation to Peersim and in relation to itself as the number of instances increase.

In the previous sections we have shown that DIPS simulation is correct, that it complies with the expected behavior set by similar simulations in Peersim. Empirical data shows that under the right configuration DIPS behaves as it would be expected from a centralized simulation.

We have also shown that the implementation, and network communication overhead while present, is manageable and as the simulation size increases, it is possible to achieve speedups. These features guarantee that DIPS correctly simulates peer-to-peer networks and its losses in performance do not hinder usability beyond reason.

This section is the culmination of all before it, now that we have a simulator prototype that is able to run correct, acceptably fast distributed simulations, we must test the simulator for its scaling capabilities which were always the primary goal of this project.

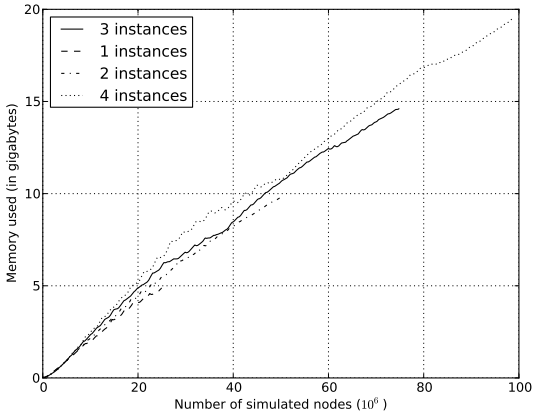

Fig. 5. Memory used by DIPS as a function of the simulated network size, using a 2000 point moving average.

The first step when initializing a simulation is to create the network, on most simulations this will be the bulk of the memory space occupied by the simulation. Simulated nodes hold information that must be stored, the sheer number of nodes in a simulated network can quickly overrun the memory limits available to the simulator. It is one of a primary goals of DIPS that any larger simulation can be run by adding the necessary number of instances to the simulator network.

In this test we show the variation on the JVM memory usage during the initialization phase of the simulation. Unlike other tests before the simulation chosen here is not important. In this test the simulation will be stopped as soon as the the network initialization phase is over.

To run the simulation we will use AWS EC2 large instances with the JVM limited to 7GB memory heap. We will runt the test on a DIPS simulator running on 1, 2, 3 and 4 instances. On each test run, we will try to initialize a network of 4 million nodes per gigabyte available to the simulator (further limited to 5GB per instance, or a total of 20 million nodes per instance).

During the initialization phase, with the initialization of every 10000 nodes we will save the memory usage in the current JVM, we will then collect these values from each instance and plot the memory used per number of simulated nodes.

We expect that the growth of memory usage by number of nodes is linear, we also expect that the memory overhead of the distributed simulator to be negligible when compared with the size of the simulated network.

The results are displayed in Figure 5. In the figure we see a moving average of the memory used by the JVM during each instant of the initialization phase.

It is important to note that the only component of the simulation that was left out of this test is message queue. Every other simulator component is included in this test and we can see that the overhead of a distributed simulator not only is negligible but appears to be non existent.

Results are extremely satisfactory, the memory usage growth is linear with the growth in network size, following an expression:

$$\alpha x + \beta$$

The observable results show that $\beta \approx 0$, indicating a simulator overhead of 0, and even more interesting $\alpha < 1$ which indicates that memory usage grows slightly slower than the network size. Although this could be attributed to compiler optimizations, we believe it is JVM runtime that is more liberal with memory allocation when there is a great amount of free memory, and becomes more strict when the available memory is becoming scarce.

6 Conclusion

In this document, we addressed the simulation of peer-to-peer overlay protocols to assist the design, development and evaluation of peer-to-peer infrastructures and applications. These have had historical importance in the development of current network aware applications. In fact, when a developer creates a peer-to-peer protocol, even if analytically deemed as correct, efficient and scalable, he needs a test environment to evaluate the protocol's characteristics. As peer-to-peer protocols are usually designed to connect a very large number of nodes, a simulation environment is necessary to convincingly test the protocol. As it is expected that the number of network connected devices will grow exponentially, peer-to-peer applications will become ever-more relevant, and a scalable and fast simulation even more needed.

During this work, we started by studying the state of the art of peer-to-peer simulation, in order to point out its shortcomings. We found that current peer-to-peer simulation suffers from a peer-count limitation due to memory usage, that cannot be overcome on a single computer. Other approaches, such as a virtualized deployment environment, have shown to be

inefficient being unable to execute simulation at an acceptable speed. To address this, we defended the need for a custom made solution aware of peer-to-peer simulation implementation characteristics, able to remove the memory limit and still execute the simulation with acceptable performance.

We propose DIPS, a Distributed Implementation of the Peersim Simulator, which is, as the name indicates is an extension to the Peersim simulator to take advantage of distributed resources, both memory and CPU. We took those concepts that are the basis of Peersim, and extended them so that they can adequately used in a distributed context. We aimed at guaranteeing that losses in performance of the simulation, due to the new distributed characteristics were minimized. The development was carried out using Java and Scala, and additional mechanisms such as load balancing, checkpointing, migration were implemented.

We evaluated this work regarding both qualitative as well as as quantitative aspects. First, we investigated whether the expected properties of simulated protocols were upheld. Then, we addressed scalability and performance regarding the memory barrier and possible speed-ups. We took into account network churn, and measure the costs and benefits of: coordination of several instances, message bundling, bounded divergence. This evaluation aimed at metrics such as the number of local events processed at each instance, deviation in latency regarding local and remote messages, message bundling, memory occupation. Results are globally encouraging and this work is able to circumvent the current major limitation, memory usage and peer-count in simulations, allowing larger and more realistic simulations of novel peer-to-peer overlay protocols.

6.1 Future Work

In the future we would like to address some open issues:

- Development of a topology modeling language or templates that helps the design of protocols and interoperability between simulators.
- Further benchmarking of DIPS with quantitative measurements on the measure of compliance with sequential simulations.
- Evaluation of the speed speed of convergence of simulated protocols, compared with sequential simulation.
- Simulation scheduling with resource awareness dealing with instances running on asymmetric machines or machines with variable load.

References

1. Montresor, A., Jelasity, M.: Peersim: A scalable p2p simulator. In: Peer-to-Peer Computing, 2009. P2P'09. IEEE Ninth International Conference on, IEEE (2009) 99–100
2. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. SIGCOMM Comput. Commun. Rev. **31** (August 2001) 149–160
3. Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems (2001)
4. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content-addressable network. SIGCOMM Comput. Commun. Rev. **31** (August 2001) 161–172
5. Zhao, B.Y., Kubiatowicz, J., Joseph, A.D., Zhao, B.Y., Kubiatowicz, J., Joseph, A.D.: Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical report (2001)
6. Maymounkov, P., Mazières, D.: Kademlia: A peer-to-peer information system based on the xor metric (2002)
7. Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., Zhao, B.: Oceanstore: An architecture for global-scale persistent storage. (2000) 190–201
8. Adya, A., Bolosky, W.J., Castro, M., Cermak, G., Chaiken, R., Douceur, J.R., Jon, Howell, J., Lorch, J.R., Theimer, M., Wattenhofer, R.P.: Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In: In Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI. (2002) 1–14
9. Castro, M., Druschel, P., Kermarrec, A.M., Rowstron, A.: Scribe: A large-scale and decentralized application-level multicast infrastructure. IEEE Journal on Selected Areas in Communications (JSAC **20** (2002) 2002
10. Gupta, A., Sahin, O., Agrawal, D., Abbadi, A.: Meghdoot: Content-based publish/subscribe over P2P networks. In: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware, Springer-Verlag New York, Inc. (2004) 254–273
11. Gil, T., Kaashoek, F., Li, J., Morris, R., Stribling, J.: p2psim, a simulator for peer-to-peer protocols (2003)
12. Shudo, K., Tanaka, Y., Sekiguchi, S.: Overlay weaver: An overlay construction toolkit. Computer Communications **31**(2) (2008) 402–412
13. Garcia, P., Pairot, C., Mondéjar, R., Pujol, J., Tejedor, H., Rallo, R.: Planetsim: A new overlay network simulation framework. Software Engineering and Middleware (2005) 123–136
14. Dabek, F., Zhao, B., Druschel, P., Kubiatowicz, J., Stoica, I.: Towards a common api for structured peer-to-peer overlays. (2003)

15. Medina, A., Lakhina, A., Matta, I., Byers, J.: BRITE: An approach to universal topology generation. In: Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2001. Proceedings. Ninth International Symposium on, IEEE (2002) 346–353
16. Naicken, S., Basu, A., Livingston, B., Rodhetbhai, S.: A survey of peer-to-peer network simulators. In: Proceedings of The Seventh Annual Postgraduate Symposium, Liverpool, UK, Citeseer (2006)
17. Jennings, N., Wooldridge, M.: Applications of intelligent agents. Agent technology: Foundations, applications and markets (1998) 3–28
18. Hanks, S., Pollack, M., Cohen, P.: Benchmarks, test beds, controlled experimentation, and the design of agent architectures. AI magazine **14**(4) (1993) 17
19. Hepplewhite, R., Baxter, J.: Broad agents for intelligent battlefield simulation. In: Proceedings of the 6th Conference on Computer Generated Forces and Behavioural Representation. (1996)