INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Distributed Peer-to-Peer Simulation

Vasco de Carvalho Fernandes

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Júri

Presidente:	Prof. Dr.ª Ana Maria Severino de Almeida e Paiva (DEI)
Orientador:	Prof. Dr. Luís Manuel Antunes Veiga (DEI)
Co-Orientador:	Prof. Dr. João Coelho Garcia (DEI)
Vogal:	Prof. Dr. João Paulo Carvalho (DEEC)

Outubro de 2011

Acknowledgments

I would like to express my gratitude to my supervisor, Prof. Dr. Luís Veiga whose unlimited patience and confidence in my abilities made this thesis possible. I would also like to thank my co-supervisor Prof. Dr. João Garcia, who has always set an amazing standard of professionalism and dedication.

I must also acknowledge Prof. Dr. Luísa Coheur that has has always selflessly supported me when I needed it and whose kindness has left an indelible mark in my path as student.

A special thanks to Eng. João Trindade whose door was always open, Eng. Tiago Picado who taught me stars are not so far away and António Novais for his friendship. One very special thank you to Frederico Gonçalves without whom this thesis would not have been.

A grateful acknowledgment to INESC-ID Lisboa, Instituto Superior Técnico and Fundação para a Ciência e Tecnologia.

This thesis is dedicated to my family, with profound gratitude. Specially to my father who kept believing in what at times seemed an endless pursuit, to his great sacrifice.

— Finally, to my better half Telma, who has always believed in me, sometimes far beyond reason. Who is a just part of me has I am myself.

Abstract

Peer-to-peer applications and overlays are very important in current day-to-day applications. These applications bring numerous benefits such as decentralized control, resource optimization and resilience. Simulation has been an indispensable tool for researchers and academics to evaluate their work. As current applications move to a more distributed model, peer-to-peer simulation will take a front seat in innovation and research.

Current peer-to-peer simulators are flawed and unable to fully serve their purpose. Limitations in memory and performance of a single machine are two restrictive for modern distributed models. We propose DIPS, a distributed implementation of the Peersim simulator to overcome these limitations. We define a wrapper around the Peersim concepts, to bring the simulator out of the restrictions of single machine deployments to the limitless scalability of a distributed system. The new simulator must be correct, fast and have a low memory footprint.

We propose an architecture for DIPS, identify possible pitfalls and propose solutions to help DIPS achieve simple, effective, unbounded scalability. We build and evaluate a DIPS prototype as a proof of concept.

Contents

List of Figures

List of Tables

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.

1.1 Background

Network communication architectures defined as peer-to-peer are the basis of a number of systems regarding sharing of computer resources (cycles, storage, content), directly between endpoints without an intermediary.

Applications base themselves on a peer-to-peer architecture, primarily due to its capacity to cope with an ever changing composition of the network 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.

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.

Current peer-to-peer simulators are incomplete tools. They bound developers by limiting the type and broadness of simulations that can be performed. Memory limitations, complexity of APIs, poor performance, are some of the problems that plague the peer-to-peer simulator domain.

1.2 Goal Statement

In this thesis we will create a specification of a distributed peer-to-peer simulator. The simulator must be able to fulfill the following requisites:

Adequacy The simulator must adequately perform peer-to-peer simulation. This means it should provide a simulation environment that adequately simulates peer-to-peer networks behavior once deployed.

Performance The distributed nature of the simulator cannot result in a unrecoverable performance hindrance. Adding instances to the distributed simulation should eventually provide a speedup when compared to a single instance simulator.

Scalability There should not be a memory limit to the simulation in the distributed simulator. Adding new instances to the simulation should increase the total amount of available memory allowing the simulation of networks of ever increasing size.

1.3 Document Structure

This document is divided in four main chapters. In the chapter 2 we discuss the state of the art on peer-to-peer, centralized, parallelized, distributed simulation, simulation of peer-to-peer and agent based simulation. On chapter 3 chapter we present DIPS, a distributed peer-to-peer simulator, and give a detailed view of its architecture. On chapter 4 we discuss the implementation details of the DIPS prototype. Chapter 5 evaluates the adequacy, performance and memory overhead of the DIPS prototype.

2 State of the Art

2.1 Peer-to-peer

We will look at the historical and established peer-to-peer protocols and their underlying architectures, we will also look at current peer-to-peer systems implementations, both commercial and academic. We will also look at peer-to-peer systems built or adapted to provide resource discovery mechanisms.

Throughout literature peer-to-peer varies in its definition particularly when considering the broadness of the term. The strictest definitions only count as peer-to-peer systems, truly decentralized systems where each node has exactly the same responsibility as any other node. This definition leaves out some systems commonly accepted as peer-to-peer, such as Napster or even the Kazaa, which are responsible for a great share of the popularity and wide-spread use of peer-to-peer technologies.

A more broad definition and widely accepted is “peer-to-peer is a class of applications that take advantage of resources, storage, cycles, content, human presence—available at the edges of the Internet”. This definition encompasses all applications that promote communication between independent nodes, i.e. nodes that have a “will” not dependent on the well being of the network in our interpretation of “the edges of the Internet”.

There are two defining characteristics of peer-to-peer architectures:

Decentralized core functionality peers engage in direct communication without the intermediation of a central server. Centralized servers are sometimes used to accomplish or help accomplish certain tasks (bootstrapping, indexing and others). Nodes must take action regarding organization as well as other application specific functionality.

Resilience to churn high churn (peers leaving and joining the network) must be the normal network state, stability must be maintained after and during peers joins and leaves, be it voluntary or due to failure.

2.1.1 Protocols

Napster

Napster was a file sharing utility that is commonly seen as the birth of peer-to-peer applications. Although it was not itself peer-to-peer network overlay, it did not have a notion of network organization, it introduced the idea of peers communicating with each other without the mediation of a server. It was also the demise of Napster in court, brought down because of its single point of failure, that inspired the distributed routing mechanisms that we associate today with peer-to-peer protocols.

Napster allowed users to share their own files and search other users’ shared files. It used a central server to:

- Index users.
- Index files.
- Perform filename searches.
- Map filenames to the users sharing them.

Actual transfer of files between users was done in a peer-to-peer fashion.

Unstructured Protocols

Unstructured peer-to-peer protocols organize the network overlay in a random graph. The purpose of the network overlay is to provide an indirect link between all nodes so that access to all data in the network is theoretically possible, from any point in the network without the need for centralized servers. The position of a node in the network overlay is generally determined by the bootstrap node, either explicitly or implicitly. Queries are not guaranteed to return all or even any results, however this best effort approach allows a minimal reorganization of both the network overlay and the underlying data, under high churn. Unstructured peer-to-peer protocols are generally tied to their applications, this is the case of Gnutella [46] and FastTrack [28] which we will look in more detail. Freenet [14] will be studied as well.

Freenet

Freenet is a peer-to-peer key-value storage system built for anonymity. Keys are generated by a combination of the SHA-1 hashes of both a short data descriptive text and the users unique namespace.

Peers in the Freenet's underlying network overlay only know their immediate neighbors. Requests are issued for a given key, each node chooses one of its neighbors and forwards the request to that neighbor. Requests are assigned a pseudo unique identifier, guaranteeing that a request does not loop over the same subset of nodes. Nodes must reject requests already forwarded. A request is forwarded until either it is satisfied or it exceeds its Hops-to-Live limit, the maximum number of times a request may be forward between nodes. The routing algorithm improves over time by keeping track of previously queries. Thus, the algorithm performs best for popular content.

Gnutella

Gnutella is a decentralized protocol that provides distributed search. Unlike Freenet searches in Gnutella may return multiple results, therefore requests are forwarded using a flooding mechanism. This design is very resilient even under high churn, however it is not scalable [30]. Like in Freenet, request and response messages are uniquely identified as to prevent nodes to forward the same message more than once. Messages must also respect a predefined Hops-to-Live count.

FastTrack

FastTrack is the protocol behind the filesharing application Kazaa. It provides a decentralized search service able to perform queries on file meta-data. FastTrack utilizes the concept of super-peers, unlike Gnutella (the original version) and Freenet, not all peers have the same responsability. Nodes with high bandwidth, processing power and storage space may volunteer to be super-peers. These special nodes cache metadata from their neighbor peers improving the query process by centralizing all their information. The network still works without super-peers and if one fails, another one is elected. The FastTrack network is therefore a hierarchical network where most of the queries are preformed at the high performance super-peers level, and the communication between low level peers serves only to maintain the network status, i.e. handle churn, handle content modification and transfer file contents.

Unstructured peer-to-peer protocols organize nodes in a network in order to guarantee communication. A request originating anywhere in the network, given enough time and resources, will arrive at its destination(s). However, in practical situations requests are limited to a level of locality by their Hops-to-Live limit.

Structured Protocols

Structured peer-to-peer protocols offer two major guarantees:

- A request will reach its destination. And as a corollary, if an object is present in the network, it can be found.
- The number of hops a request must perform to reach its destination is bounded.

Chord

Chord [52] is a peer-to-peer lookup protocol that builds on the concept of a distributed hash table (DHT) to provide a scalable, decentralized key-value pair lookup system over peer-to-peer networks. It uses query routing to satisfy the lookup and is bounded to $O(log(n))$ hops for any query. Simplicity is a key feature, Chord supports only one operation: given a key, it maps that key onto a node.

Chord also proposes to overcome limitations of semi-centralized peer-to-peer applications and unstructured peer-to-peer protocols. Such as:

- A central server as a single point of failure (Napster).
- The number of messages to satisfy a query increases linearly with the number of nodes in the system (Gnutella).
- Even though minimized, availability problems are not solved by the use of super-peers.

Chord has five fundamental properties:

Decentralization All nodes have the same role, no node is more important or has greater responsibility than other nodes.

Availability Nodes responsible for a key can always be found even during massive leaves or joins.

Scalability Lookup is bounded to $O(log(n))$, therefore network size has little effect on query speed.

Load balance Chord uses a consistent hash function that guarantees a key responsibility to be evenly spread across the network.

Flexible naming Chord does not impose constraints on key structure.

Chord uses a consistent hash function (SHA-1) to guarantee that the key space is spread evenly across the network. The network is defined as circular linear identifier namespace called the Chord ring. The identifier is a m-bit number, where m is chosen before the setup of the network. Both key names and node names are translated into to this name space using the SHA-1 hash function.

Nodes have positions on the ring directly defined by the numerical ordering of their identifiers. Nodes only know the location of their direct successors, a node's successor is:

- the node whose identifier is the smallest number, larger than the current nodes identifier.

- or, if the previous condition is not possible, the node whose identifier is the smallest number of all nodes.

successor is the lookup function that uses successor information on the current node to get *closer* to the key location; a key location is the node whose identifier is smallest number, larger than the keys identifier (same process than the nodes successor).

In order to accelerate lookup, Chord proposes an optimization, the **Finger Table**. Each nodes stores the location of m (as defined before) nodes according to the following formula:

$$finger[i] = successor((n + 2^i - 1)\%2^m)$$

To ensure that correct execution of lookups as the nodes leave/join the network, Chord must ensure that each node's pointer is up to date. The *stabilize* function is called periodically on each node to accomplish this. The function asks for the current node's successor predecessor, which should be the current node unless a new node as joined; if a new node has joined, the pointer is updated and new successor notified.

Pastry

Pastry [47] is a scalable distributed object location and routing middleware for wide-area peer-to-peer applications. It provides application level routing based on a self-organizing network of nodes connected to the Internet.

The Pastry network is composed of nodes, each one with a unique identifier $nodeId$. When provided with a message and a key, Pastry routes the message to the node with the $nodeId$ numerically closer to that key. A Pastry node routes messages to the node with the $nodeId$ numerically closer to its own. $nodeId$ and key are numbers abstracted as a sequence of digits in base 2^b.

When routing of messages to nodes, based on a key, the expected number of routing steps is $O(log(n))$, where n is the number of nodes in the network. It also provides callbacks to the application during routing. Pastry accommodates network locality, it seeks to minimize the messages travel distance according to some metric such as the number of IP routing hops or the latency in connections. Each node keeps track of its immediate neighbors in the $nodeId$ space, callbacks for the application are provided for node arrivals, failures and recoveries.

In order to route a message, a given node chooses one of its neighbors, which should have a prefix (or b bits) closer to the message key, that is if the current $nodeId$ has a prefix with m digits in common with the key, the chosen node should have a prefix with, at least, $m + 1$ nodes in common with the key. If no such node exists, then the message is forward to a node with a $nodeId$ that has a prefix with m digits in common with the key, as long as that $nodeId$ is numericaly closer to that key.

Applications have been built using Pastry, such as a persistent storage utility called PAST [48] and a scalable publish subscribe system called SCRIBE [10] .

Content Addressable Network

Content Addressable Network [44] (CAN) is a distributed Internet-scale, hash table. Large-scale distributed systems, most particularly peer-to-peer file sharing systems such as Napster and Gnutella, could be improved by the use of a CAN.

Semi-centralized peer-to-peer applications such as Napster have problems scaling and are vulnerable (single point of failure).

Decentralized unstructured peer-to-peer protocols are only complete (all objects in the network can be found) in very small networks. As networks get bigger some objects become unreachable, so we can say unstructured peer-to-peer protocols cannot scale with respect to completeness.

CAN's first objective was to create a scalable peer-to-peer system. An indexing system used to map names to locations is central to the peer-to-peer system. The process of peer-to-peer communication is inherently scalable, the process of peer location is not. Hence the need for a scalable peer-to-peer protocol.

CAN resembles a hash table; insertion, **lookup** and deletion of (key, value) pairs are fundamental operations. It is composed of many individual nodes. Each node stores a chunk of the hash table (called a zone), as well as information about a small number of *adjacent* zones. Requests are routed towards the node whose zone contains the key. The algorithm is completely distributed (no central control or configuration), scalable (node state is independent of the systems size), it is not hierarchical and it is only dependent of the application level (no need for transport OS operating system layer integration).

Large-scale distributed systems are one possible application of the CAN protocol. These systems require that all data be permanently available and therefore an unstructured protocol would be unsuitable as basis for such systems (see section 2.1.2). Efficient insertion and removal in a large distributed storage infrastructure and a scalable indexing mechanism are essential components that can be fulfilled with CAN.

A wide-area name resolution service (a distributed non hierarchical version of DNS) would also benefit from this CAN.

Tapestry

Like Pastry, Tapestry [58] shares similarities with the work of Plaxton, Rajamaran and Richa [43].

Tapestry supports a Decentralized Object Location API [15]. The interface routes messages to endpoints. Resources are virtualized since the endpoint identifier is opaque and does not translate any of the endpoint characteristics, such as physical location.

Tapestry focus on high performance, scalability, and location-independence. It tries to maximize message throughput and minimize latency. Tapestry exploits locality in routing messages to mobile endpoints, such as object replicas. The author claims that simulation shows that operation succeed nearly 100% of the time, even under high churn. This, however, has been disputed [45].

The routing algorithm is similar to Pastry, messages are routed to a node that shares a larger prefix with the key for that message.

Like Pastry, Tapestry builds locally optimal routing tables at initialization and maintains them. Using a metric of choice, such has network hops, the relative delay penalty, i.e. the ratio between the distance traveled by a message to an endpoint and the minimal distance is two or less in a wide-area.

Tapestry uses multiple roots for each data object to avoid single point of failure.

Examples of applications built with Tapestry are Ocean Store [27] and Bayeux [59].

Kademlia

Kademlia [35] is a peer-to-peer distributed hash table. It differs from other structured peer-to-peer protocols as it tries to minimize the number of configuration messages. Configuration is organic, it spreads automatically with key lookups. Routing is done through low latency paths. Opaque keys of 160-bit are used, key/value pairs are stored on nodes with id closest to the key. It utilizes a XOR metric to measure distance between points in a key space, the distance between x and y is $x \oplus y$. Symmetry in XOR allows queries to be forward through the same nodes already present in the destinations routing table. Kademlia treats nodes as leaves in a binary tree with the node's position determined by the shortest unique prefix of its id. The protocol guarantees that each node knows of a node belonging to each of the sub-trees not containing this node.

Viceroy

Viceroy [32] is another DHT system that employs consistent hashing. Its structure is an approximation of a butterfly network. The number of hops required to reach a node is bounded with high probability to $O(log(n))$ and the number of nodes each node must maintain contact is seven. This constant link number makes churn less burdensome as the number of nodes affected by the arrival and departure of any given node is lowered.

Koorde

"Koorde is a simple DHT that exploits the Bruijn graphs[5]" [25]. Koorde combines the approach of Chord with the Bruijn graphs, embedding the graph on the identifier circle. As a result Koorde maintains Chords $O(log(n))$ max hop bound, but, like Viceroy, requires only a constant degree, the number of neighbors a node must maintain contact with. Unlike Viceroy the number of hops is bounded to $O(log(n))$

Symphony

Symphony [33] is yet another example of a DHT. It is inspired by Kleinbergs's Small World [26]. Like both Koorde and Viceroy it requires only a $O(1)$ degree. The max hop bound is $O(\frac{1}{k}log^2(n))$. However, Symphony allows a trade off between the degree of the network and the max hop bound at runtime.

2.1.2 Systems

OceanStore

OceanStore [27] is a distributed storage system. Data is stored, replicated, versioned and cached over a peer-to-peer network. It was designed with two differentiating goals:

1. Support for **nomadic data**. Data flows through the network freely, due to the need for data locality relative to its owner. Data may cached anywhere, at anytime.
2. The ability to be deployed over an **untrusted infrastructure**. OceanStore assumes the infrastructure is untrusted. All data in the infrastructure is encrypted. However, it participates in the protocols regarding consistency management, so servers are expected to function correctly.

	Simulation parameters	Network width	Network degree	Locality properties
Chord	n: number of peers	$O(log(n))$	$log(n)$	None
Pastry	n: number of peers; b: base of the chosen identifier	$O(log_b(n))$	$2b * log_b(n)$	Accounts for locality in routing
CAN	n: number of peers; d: number of dimensions	$O(d.n^{\frac{1}{d}})$	$2d$	None
Tapestry	n: number of peers; b: base of the chosen identifier	$O(log_b(n))$	$log_b(n)$	Accounts for locality in routing
Kademilia	n: number of peers; b: base of the chosen identifier; c: small constant	$O(log_b(n)) + c$	$b * log_b(n) + b$	Accounts for latency when choosing routing path
Viceroy	n: number of peers;	$O(log(n))$ with high probability	$O(1)$	None
Koorde	n: number of peers;	$O(log(n))$	$O(1)$	None
Symphony	n: number of peers; k constant	$O(\frac{1}{k}log^2(n))$	$O(1)$	None

Table 2.1: Comparison of structured peer-to-peer protocols.

Object location through routing is done using a two tier approach. First a distributed algorithm based on a modified version of a Bloom filter, will try locate the object. Since this is a probabilistic solution it may fail. In case of failure the object will be located using a version of the Plaxton algorithm [43]. Replica placement is published on the object's root, i.e. the server with $nodeId$ responsible for the object's id.

Squirrel

Squirrel [23] is a decentralized peer-to-peer web cache. It uses Pastry as its object location service. Pastry identifies nodes that contain cached copies of a requested object. Squirrel may operate using one of two modes. Following a request, a client node will contact the node responsible for that request, the home node:

1. If the home node does not have the object, it will request it from the remote server and send it to the client
2. The home has a directory, potentially empty, with references of other nodes that may have a copy of the object. These were created at previous requests. A randomly chosen reference is sent back to the client, and he is optimistically added to the directory.

Evaluation of the Squirrel system was performed using a mathematical simulation, fed with real data acquired by executing two different traces of Internet usage. Ranging from 105 to 36782 clients.

Scribe

Scribe [10] is an application-level multicast infrastructure built on top of Pastry. Scribe overcomes lack of wide-spread deployment of network level multicast by building a self organizing peer-to-peer network to perform this task.

Any Scribe node can create a group. Other nodes may join that group. The system provides a best-effort delivery policy, and no delivery order guarantee. Each group has a $groupId$, and information of the nodes in the group. These are mapped into a key, $message$ pair. The Pastry node responsible for the $groupId$ acts as a *rendez-vous* point, for that group. It is also possible to force the *rendez-vous* point to be the group creator. Message delivery is done through a multicast tree algorithm similar to reverse path algorithm [16].

Scribe was evaluated using a custom build discrete event simulator. The simulation was composed of 100,000 nodes. The simulation was composed of both the Pastry nodes and the underlying routers (5,050), this allowed to simulate delay penalty of application multicast over network multicast.

PAST

PAST [48] is a peer-to-peer persistent storage system not unlike OceanStore. Files are stored, cached and replicated over a network of peer-to-peer nodes organized using the Pastry protocol. Files stored in PAST possess a unique id and are therefore immutable. PAST uses Pastry's network locality to minimize client latency.

PAST evaluation was done using a custom simulation over the actual implementation of the system. It used a single Java virtual machine. Simulation was fed data from two traces, one referencing 4,000,000 documents and the other 2,027,908.

Meghdoot

Meghdoot [19] is a publish subscribe system based on CAN. The events are described as tuple of attributes where each attribute has a name and, a value or range. Subscriptions are stored in the network. When an event arrives, the network must identify all matching subscriptions and deliver the event.

Simulation of Meghdoot was done using a custom simulator. Two event sets were used, one generated randomly, the other real stock data. Subscriptions were generated randomly. The event sets contained 115,000 objects and 115,353 respectively. The system was tested with 100, 1000 and 10,000 peers.

Others

Other examples of peer-to-peer systems are Ivy [39], a versioned file storage system. Farsite [1] is another distributed file storage system.

2.1.3 Peer-to-Peer protocols for Resource Discovery

Nodes participating in a network usually share resources between them. The systems we have seen so far have these resources completely specified and integrated in the underlying protocol, namely files, documents or even topics. Grid like networks can be built on top of a peer-to-peer overlay only if the overlay is capable of providing a resource discovery service for computing resources (i.e., CPU time).

It has been argued in literature that Grid and Peer-to-Peer systems will eventually converge [54].

Resource discovery protocols in peer-to-peer systems can be devided as targeting structured and unstructured networks. Examples of these protocols for unstructured networks can be found in [22, 34, 55].

Resource discovery in unstructured peer-to-peer networks

Regarding architecture, nodes are generally organized into a cluster, mostly grouped by virtual organization, where one or more of the nodes act as a super-peers.

Resource indexing is done at the level of the super-peer or equivalent, or, in Iamnitchi et al. [22] each peer maintain information about one or more resources.

Query resolution is done using a routing index. Statistical methods based on previous querys select the super-peers with the highest probability of success. However, in Iamnitchi et al. queries are routed using either random walk or a learning-based best-neighbor algorithm.

Experiments [34] show that the super-peer model is an appropriate model for grid like services, due to its closeness to the current Grid model.

Resource discovery in structured peer-to-peer networks

MAAN [8] proposes an extension to the Chord protocol to accept multi-attribute range-queries. Queries are composed of multiple single attribute queries, one different DHT per attribute.

Andrzejak et al. [3] extended the CAN system to support range queries. Resources are described by attributes. Queries on discrete attributes will be routed using regular CAN functionality, queries over continuous spaces will use the extension. As in MAAN, there is one DHT per attribute.

SWORD [41] uses a DHT system called Bamboo, similar to Pastry. SWORD provides mechanisms for multi-attribute range queries as before. However in SWORD each attribute is assigned to a subregion of a single DHT.

XenoSearch [51] extends Pastry. Once again each attribute is mapped to its own Pastry ring. Attribute range queries are preformed separately and then combined through intersection.

Mercury [4] is based on Symphony. Each single attribute is assigned a different DHT. Each node stores all information on all attributes on all hubs. This way the smallest range query is chosen and therefore only one DHT needs to be queried.

2.2 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/abstraction levels in their configuration and execution according to the level of abstraction desirable for each type of simulation.

We will look at the simulation of systems, networks and agents, and their relevance to the distributed simulation of peer-to-peer network overlays. We will look at two types of simulation: discrete-event and real-time simulation.

Discrete-event Simulation

Traditional discrete-event simulations are executed in a sequential manner. A variable *clock* maintains the current status of the simulation and is updated it progresses. A *eventlist* data structure holds a set of messages scheduled to be delivered in the future. The message with the closer delivery time is removed from the event list, the corresponding process is simulated and the *clock* is updated to the delivery time. If the simulated process generates more messages, these are added to the event list. This is called event-driven simulation because the clock always moves to the next delivery time and never in between.

Real-time Simulation

Real-time simulation evolved from virtual training environments. Particularly useful to the military, it respects real-time to integrate simulated components with live entities, such as humans. It suffers from scalability problems, as the whole simulation and simulated activities must be executed in real-time (probably in concurrent manner).

2.2.1 Network Simulation

Network simulation is a low level type of simulation. Network simulation tools model a communications network by calculating the behavior of interacting network components such as hosts and routers, or even more abstract entities such as data links. Network simulation tools allow engineers to observe the behavior of network components under specific conditions without the deployment costs of a real large-scale network.

High-level design problems for the digital communication infrastructure are very challenging. The large scale and the heterogeneity of applications, traffic and media, combined with QoS restrictions and unreliable connectivity, makes this a non-trivial problem.

Application and protocol development at the network level involve a number of heterogeneous nodes that are both expensive and hard to assemble. Simulation is therefore the best solution when testing low level network applications and protocols.

Ns-2

Ns-2 [1] is a discrete-event network simulator. Ns-2 is the *defacto* standard tool for network simulation.

NS-2 generates data down to the packet level. The simulator ships with a number of simulated protocols such as *udp* and *tcp*. The modular approach allows for the implementation of custom protocols, this can be done by extending base classes of the simulator.

Simulations are executed and controlled through configuration scripts written in the OTcl language with a custom API.

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.

The peer-to-peer simulation as in general network simulation, is composed of two different pieces of code. The simulator code is responsible for the execution of the simulation, it creates peers, maintains the main simulation loop and delivers messages if necessary. The simulated protocol code is responsible for the logic particular to the protocol, it is the code to be run when a node needs to be simulated. This code will be run either to simulate message arrival or at regular interval during the main loop.

We will look at current peer-to-peer simulators regarding their:

- Simulation type
- Scalability
- Usability
- Underlying network simulation fidelity.

Current peer-to-peer simulators may offer two modes of operation, the event-driven mode is a discrete-event simulation closely related to more general network simulation and to the simulation of systems. Messages are sent between simulated peers, they are saved in a queue and processed in order by the simulation engine, that runs code to simulate the destination peer receiving the message.

The other type of simulation is a cycle-based simulation, it resembles real-time 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

[1] http://www.isi.edu/nsnam/ns/

offers a greater abstraction than the event-based engine as the simulated peers information are available at all points of the simulation. The level of encapsulation when simulating an individual peer is left to the implementor of the protocol to decide.

Simulation of very large networks is particularly relevant when simulating peer-to-peer protocols and systems. The usual deployment environment for a peer-to-peer application is a wide-area network. Whether a peer-to-peer simulator can scale to the size of real wide-area network is a very important factor in choosing a peer-to-peer simulator.

Another important factor is how well documented is the simulator. The simulator must be configured using a configuration language that is either declarative or procedural, we must take into consideration how easy and powerful it is.

Peersim

Peersim [37] 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.

The cycle-based mode is well documented with examples, tutorials and class level documentation. The event-driven mode however, is only documented at class level. Peersim utilizes a simple custom language for the simulation configuration. All control and statistical gathering must be done by extending classes of the simulator that will then be run in the simulation.

Peersim offers some underlying network simulation in the event-driven mode, it will respect message delay as requested by the sender.

P2PSim

P2PSim [18] 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 [50] 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 [17] is also a discrete-event simulator written in Java. It uses the Common API given in [15].

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 [36] information for this purpose.

2.2.2 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.

Real-time simulation is typically parallel as components should be simulated concurrently given the real-time restrictions and the interaction with live components. In real-time simulation even if some components are implemented sequentially, partition for parallel execution is a trivial process since all events must be made available to all (interested) components at the time they occur.

Discrete event simulation is usually sequential.

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 de-synchronization.

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.

Simulator	Engine Type	Scalability	Usability	Underlying Network
PeerSim	cycle-driven and discrete-event	1,000,000 nodes	good documentation for the cycle-driven engine	not modeled
P2PSim	discrete-event	3,000 nodes	some documentation	strong underlying network simulation
Overlay Weaver	discrete-event	4,000 nodes (limited by OS max threads)	appropriate documentation	not modeled
PlanetSim	discrete-event	100,000 nodes	good documentation	some support

Table 2.2: Comparison of Peer-to-Peer Simulators

Discrete-event system simulations are by their very nature sequential. Unfortunately this means existing simulations cannot be partitioned for concurrent execution.

Sophisticated clock synchronization techniques are required to ensure cause-effect relationships.

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.

Parallelization of a discrete-event simulation can be approached using one of two strategies, regarding causality:

Conservative strategy If a process knows an event with a time stamp T_1, it can only process this event if, for all other events T_n received afterwards: $T_1 < T_n$. A parallel discrete-event algorithm was developed independently by Chandy and Mistra [11] and Bryant [6]. The simulation must statically define links between communicating processes. By guaranteeing that messages are sent chronologically across links, the process can repeatedly select the link with the lowest clock, and if there are any messages there, process it. This might lead to deadlocks when all processes are waiting on links with no messages. The problem of deadlocks can be solved using null messages, a process will send an empty message to update the links clock preventing deadlocks. This is highly inefficient so other approaches have been proposed [12].

Optimistic strategy Is based on the idea of rollback. The process does not have to respect causality in processing received messages, it may process all messages it has already received (in chronological order) independent of the incoming link clocks. To recover from errors, the process maintains a Local Virtual Time

(LVT) equal to maximum of all processed messages. It must also maintain a record of its actions from the simulation time (the lowest time stamp on all links) up to its LVT. When a message with a time stamp smaller than the LVT arrives, called a *straggler*, recovery must be done. The rollback process consists on recovering the state of the process at the time of the *straggler*. The process must also undo all messages that it might have sent. The undo process involves sending an *anti-message*. The receiving process must then initiate a rollback process up to the message it has processed before the *anti-message*. This process is called Time Warp with aggressive cancellation. Alternatively, the process might only send the *anti-message* to a incorrectly sent message M if it verifies that M is not generated up the its time stamp.

An optimistic approach places an extra burden on the protocol description, as it must describe anti-messages, which are not necessary under live deployment.

Lazy cancellation may improve performance depending on the simulated system. Studies on performance using optimistic strategies can be found in [29, 31]. An optimization to the Time Warp protocol in a system where each instance is responsible for more than one component can be found in [56].

2.2.3 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

The above are problems that all distributed systems must take into account. Other problems, depending on the type of simulation may also arise. Fault tolerance, replication, shared state, interest management and load balancing are examples of those.

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 [40].

We have to look outside of network simulation to get insights on the inner workings of distributed simulators.

Simgrid [9] is a high-level simulator for scheduling in cycle-sharing systems. GridSim [7] is also a toolkit for modeling and simulation of resource managements in grid environments. These very high level simulators capture only a small portion of the complexity in grid resource allocation and management.

Other systems such as cycle sharing systems [2, 13] implement related mechanisms as they abstract units of work to be executed in distributed environments. These, as with frameworks to help distribute systems like the PVM [53], have close ties to distributed simulation as they suffer from the same problems and implement some of the same solutions regarding the distributed aspect of their operation.

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 [24]. 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 [24]. As a result, design and implementation remain extremely experimental. However, no testbed is appropriate for all agents and environments [20].

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 [21].

JAMES, a platform for telecommunication network management with agents is an example of a system that does parallel simulation [42].

Decentralized event-driven distributed simulation is particularly suitable for systems inherently asynchronous and parallel. Existing attempts model the agents environment as a part of a central time-driven simulation engine. Agents may have very different types of access to their environment. Depending on this type of access and their own implementation they might be more or less autonomous. Given traditional agent based models, distributed simulation of agents based systems differs from other discrete-event simulation in one important aspect: usual interaction is between the agent and its current environment, there is no direct interaction between agents.

Real time large scale simulation approaches the problem of interest management [38]. An interest manager matches events with the agents that have an interest in that event. This helps the system to save resources by only making available events to the agents that actually require them. Interest expressions are used to filter information so that processes only access information relevant to them.

3 Architecture

In the state of the art we introduced the currently available peer-to-peer simulators, from the description of this simulators as well as the peer-to-peer systems it becomes apparent that current peer-to-peer simulation must be run in low capacity, and therefore, low fidelity simulators.

While peer-to-peer protocols are created to manage networks with a very large amount of peers, existing simulators are unable to create very large networks. To create networks that are realistic in the number of peers the protocol is expected to handle, is essential when using simulation to study the protocol's characteristics.

Currently available simulators are limited by the resources available in the hardware they are run on. Even on the simulator capable of generating the largest networks, extremely simple simulations are limited to about 4 million nodes per gigabyte, while on more complex simulations, limitations grow exponentially. From a performance point of view, current simulators are also lackluster. If a simulation performs a particularly expensive calculation, it simply is not possible to accelerate it past a certain point, independent of the monetary/hardware resources available to the simulation creator. The immediate answer to both of these problems is to distribute the simulation, as a distributed simulation has theoretically unlimited access to extra memory and CPU units.

Distributed simulation has existed since the beginning of simulation it self. It was not until the availability of powerful commodity hardware that new simulators regressed to a centralized model. Now that there is a widespread availability of network connectivity as well as commodity idle hardware, it becomes relevant again, to try and pool together distributed resources to scale horizontally.

Distributed Implementation of the Peersim Simulator (DIPS)

We propose DIPS, a Distributed Implementation of the Peersim Simulator, which is an extension to the Peersim simulator to take advantage of distributed resources, both memory and CPU.

As it can be seen in Figure 3.1, DIPS is a set of Peersim regular instances that run one global simulation where the simulated peers (from here on called nodes) 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 the concepts that are the basis of Peersim, extend them so they can adequately be used in a distributed context. Finally we must guarantee that losses in simulation performance, due to the new distributed characteristics, are minimized. We must also guarantee that challenges created by the distributed behavior are met in a way that does not overburdens the simulation creator.

Figure 3.2 shows the architecture of DIPS divided into three large components. In this chapter we will explain in detail each of these components.

The first two, network communication and the extension of the Peersim simulator are independent. The architecture was defined so that, even though both aspects are aware of each other, each one acts as a black box to the other.

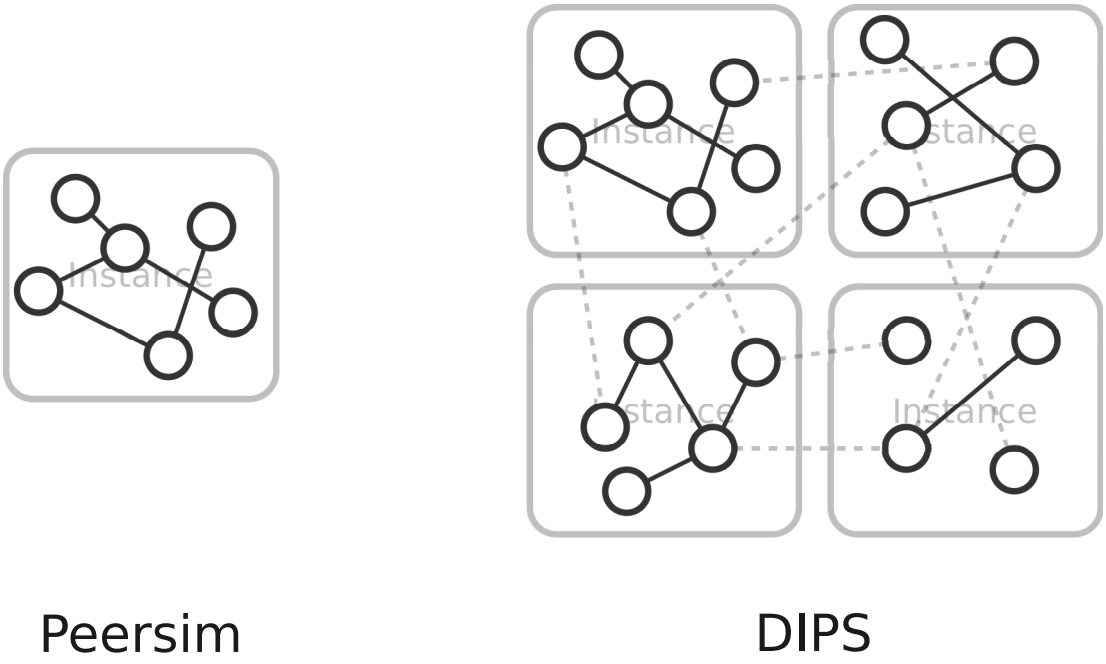


Figure 3.1: Comparison of Peersim and DIPS, simulating a peer-to-peer network

The third component, covers advanced topics regarding challenges created by the network communication between instances. This is a cross cutting component, it interacts with both other components in order ease a correct, fast execution of DIPS.

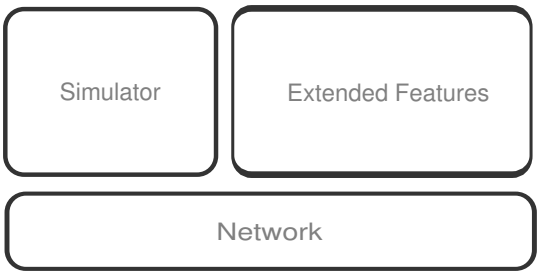


Figure 3.2: DIPS Simplified Architecture

3.1 DIPS Network Architecture

In this section we offer a bottom up view of DIPS. We start with network communication and how it is handled at the lowest abstraction. We then move on to the organization of the network from the perspective of each participant. In the last section we describe network coordination.

3.1.1 Network Communication

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. There are two main advantages to the separation between simulation and network communication.

1. Different implementations may be swapped if necessary.
2. Network communication may be run in a separate thread of control.

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.

The possibility of swapping the network communication component can be extremely useful when the medium on top of which the simulator is run, changes. There are great differences between running the simulator on top of the Internet or a Local Area Network. Ideally the simulator should be able to use the network component that better adjusts to the network conditions.

DIPS is not a parallel simulator, it does not have the synchronization mechanisms necessary to use more than one processor at the same time. This means that if one extra processing unit is available, network communication processing could be offloaded to that unit leaving the current first processor free to run the simulation uninterrupted.

Finally, by isolating network communication from simulation code and offering a well defined API to move information between instances, it becomes easier to implement DIPS, limiting the amount of changes to the simulation component in relation to the original simulator in Peersim.

The Actor Model

The actor model is a model for concurrent computation where communication is done asynchronously through message passing. An abstraction is presented in Figure 3.3. This model differs from the more widely used shared memory model, as it assumes that there is no shared memory between concurrent processes.

In this model, each process is called an actor, and is defined as an infinite loop that reacts to arriving messages. This is the design that we propose for the network communication component of DIPS.

By defining the network communication component as an actor we further help isolate it from other components in the simulator. This is not only a semantical isolation. Communication is executed through message passing therefore communication structures are clearly defined, however it is also a physical isolation, by removing shared memory from the design we guarantee independence of the simulator from the network, limiting the

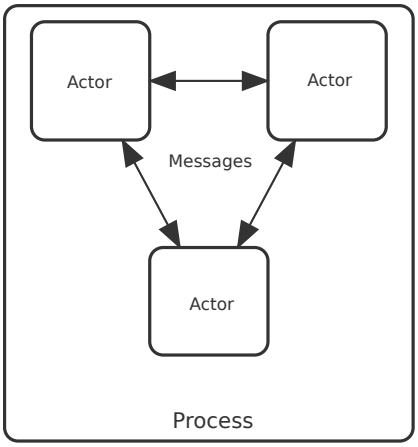


Figure 3.3: The Actor Model

impact that network communication processing can have on the performance of the simulator.

Post Office Metaphore

In the previous paragraph we have slipped slightly into the implementation, however the actor model is an important concept to understand the network communication component design.

Just like an actor, the network component is a independent component that only interacts with other components through message passing. We have called this the post office metaphor as the network component acts as a post office for the other components of the simulation.

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 post office 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.

This behavior is similar to what is expected of a post office. How messages are delivered is not the concern of the sender, only of the network component. It is also important to note that address may be physical, *i.e.* an *ip* and *port*, which would map to a street and door number in our metaphor, but addresses may also be virtual, *i.e.* a simple *ID*, which could be mapped to a P.O. box.

Subscription accomplished using a simple publish/subscribe mechanism that instead of placing the burden on the receiver to check for new messages, allows incoming messages to be delivered almost immediately after arrival. This is a mechanism that is a good fit for control components that sit idle waiting for messages. Holding

the messages until they are collected is better for components that process messages at a specific point in time, such as during an event loop.

To summarize, the network component takes care of communication for all other components in DIPS, it guarantees delivery, takes care of routing and serves as a buffer for incoming messages. It frees other components from the burdens of network communication.

3.1.2 Network Organization

So far we have seen the network component of DIPS. As the existence of this component indicates, a DIPS simulation is run in a network of DIPS instances. This network must be managed in regard to how instances join and leave, how routing is performed and, as we have seen in the previous section, the allocation of virtual addresses.

In this section we define how the network is organized, we propose two routing algorithms to manage virtual addresses, explain how the network handles edge cases that require centralized control. Also how to handle churn.

There are a few guidelines regarding the design of the DIPS network:

- The organization of the network should be simple.
- Communication should be efficient.
- Virtual address lookup must be $O(1)$.
- Organization should be versatile enough to handle a large number of instances if necessary.

Before moving on to the basic organization of the network, it is important to state the importance of virtual addresses. We will see in section 3.2 that virtual addresses are used to send messages between simulated peers, lookup of virtual to physical address will be the large majority of the network component operations, hence the need for efficiency.

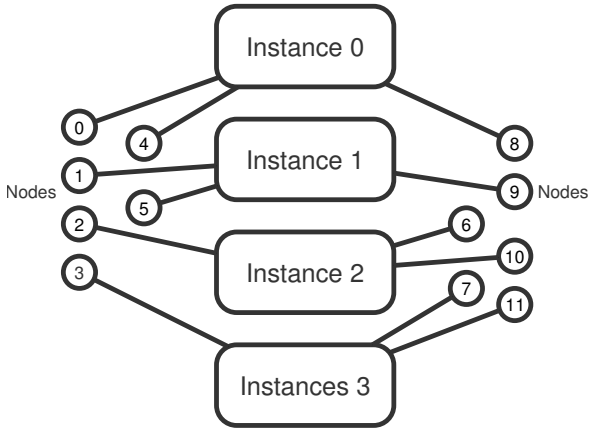


Figure 3.4: Round Robin virtual address lookup

Well-Known Organization

The most common case for the DIPS network composition will be a very small number of instances, in the order of ten. This instance will form the network before the beginning of the simulation and remain in it until the simulation finishes.

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.

Since all instances can contact all other instances, $O(1)$ virtual address lookup can be achieved simply by ordering instance *ID*'s and defining a common method of lookup to all instances. Ordering can be achieved through any number of algorithms, from alphabetical order to hashing. The lookup could use any common method of attribution, however, for reasons that will become apparent when we discuss virtual address attribution, a particularly interesting algorithm from a load balancing point of view is round robin, available in Figure 3.4.

Round robin works by translating the virtual address to a number (using an unique function) and then calculating the modulus of the ID with the number of instances in the network. The resulting number is the index of the instance, in the ordered network instance list, to which that virtual address is assign.

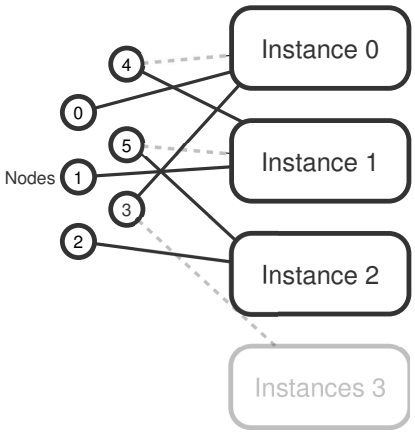


Figure 3.5: Round Robin virtual address reassignemnt in face of churn

Network Churn

Network churn is the amount of instances that join and leave the network per unit of time. Churn is important regarding virtual address allocation and asset management. When an instance leaves the network all its assets must stay in the network, and all virtual addresses assign to it must be assigned to another instance. It is part of the network architecture to handle connection and disconnection.

When an instance is disconnected either by request or as a result of failure, the network must reallocate virtual addresses so that there are no lost addresses. It is also the network responsibility to maintain communication

with the disconnecting instance in case the instance is still online, to allow other components to handle asset relocation.

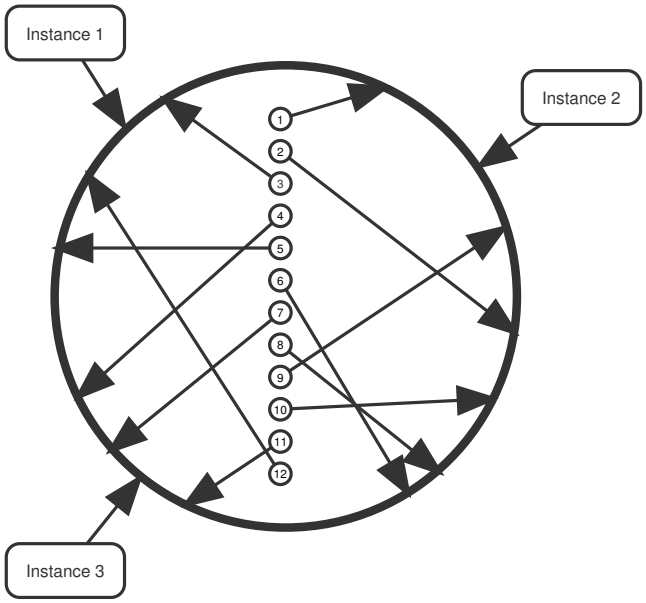


Figure 3.6: The DHT organization

Evolving to a DHT based Organization

The round robin approach presented previously, clearly does not scale. Given the expected number of DIPS instances, it makes sense for it to be the default approach. When the number of instances grows, it becomes burdensome to keep track of all instances, and it may be necessary to take a more structured approach to broadcasting.

Despite its advantages in terms of load balancing, round robin is particularly inefficient in the presence of churn as it can be seen in figure 3.5. Whenever a new instance joins the network, the number of virtual addresses that need to be remapped is:

$V * \frac{N-1}{N}$, where N is the number of instances in the network, and V is the number of virtual addresses

This is our primary concern in defining an alternative organization model for the network.

In the DHT model the network is defined as a distributed hash table where each instance position is given by the hash value of the instance's *ip* and *port* (as the *"ip:port"* string) the hash table is also extended as a CHORD like circular structure. Simulated nodes are hashed according to their global identification number, and the simulation instance responsible for any given node is defined (as in CHORD) as the instance with hash value immediately before the node's own hash value. A pseudo-code implementation of the DHT model is available in Algorithm 1.

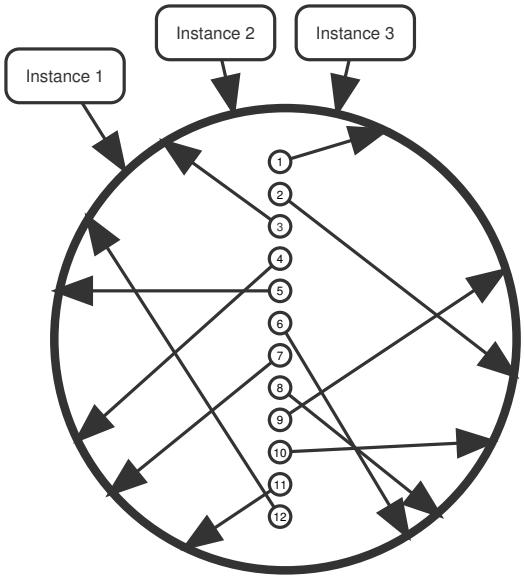


Figure 3.7: DHT Organization Problem

This model also eases the transition to a very large network, allowing future improvements in failure protocol, opening a path to the implementation of the full CHORD protocol without change to the current architecture.

At this point it is important to understand why the DHT model cannot be used from the start. Once again the relatively small number of instances plays a crucial role, while consistent hashing is expected to give a consistent division of virtual addresses per instance, this is only statistically true, as for a very small number of instances, the resulting network is likely to look like the one presented in Figure 3.7. Round robin, as a simple division of the virtual addresses per number of simulation instances not only is more efficient, but also guarantees an egalitarian distribution of the nodes independently of the number of simulation instances.

3.1.3 Token Based Coordination

Some decisions in the network must be taken from a centralized location. To make it possible, we have created a control protocol that defines the master instance in the network, the one whose decisions take priority.

The control protocol is a token based one, where only the token holder may send control messages to the network. Any instance may hold the token, however, only one instance may hold it at any given time. An instance that initiates a simulation, becomes responsible for it and may only relinquish the token when the simulation is over.

In order to acquire the token an instance must request it from its current owner. If the owner is no longer responsible for a simulation and therefore may relinquish the token, it does so immediately after the first request, responding with the new owner's ID to subsequent requests. If the current owner is forced to maintain the token in virtue of its responsibility to the simulation, it must maintain a queue of requests in the order of arrival. When the token is free to be delivered, the owner contacts each of the requesters in the queue order, until one declares

Algorithm 1 DHT routing

```
class router
        constructor: (instances) ->
                this.instances = instances

        sort: () ->
                this.instances.sort (n1, n2) ->
                        sha1(n1) < sha1(n2)

        add: (node) ->
                this.instances.push node
                this.sort()


class dht
        constructor: (local_addr) ->
                this.addr = local_addr
                this.network = new router([local_addr])

        on_new_connection: (remote_addr) ->
                this.network.push remote_addr

        route: (routable) ->
                hash = sha1 routable

                n = this.network.node.find (n) -> hash < sha1 n
```

interest in the token or the queue ends. When the token is transfered, the remaining requests in the queue are also transfered and, take priority over new requests made to the new owner.

3.2 DIPS Simulation Architecture as an Extension of Peersim

In a sentence, the simulator is medium of communication between the simulated nodes. The simulator is the core of DIPS. Even though it is its most important component, it is also the least original one.

One of the concerns when designing DIPS was to have as much compatibility with Peersim as possible, without hindering DIPS performance and scalability goals. For this reason, most of the concepts presented in this section are common to both DIPS and Peersim. It is nonetheless necessary to grasp them, in order to understand the design of the distributed simulator, as well as the more advanced concepts presented in the next section.

In this section we will go through the concepts behind a simulation, stopping to take a deeper look whenever DIPS design differs from Peersim. We will end the section describing DIPS two distributed simulation engines.

3.2.1 Peersim Overview

Peersim is a peer-to-peer simulator. Peersim is capable of simulating the behavior of large peer-to-peer networks and extract information from them. The simulated network consists of nodes, single, small footprint objects that implement part or all of the behavior of a real peer-to-peer node.

Peersim uses an engine to execute a simulation. There are two distinct types of simulation engines in Peersim, the cycle based simulation engine and the event based simulation engine.

Cycle based simulation consists in repeatedly handing over control to nodes, which in turn alter their state in response to their environment. A cycle is the period necessary for all nodes to hold control exactly one time.

Event based simulation abstracts the concept of the cycle based simulation, substituting direct access to nodes with message passing, this abstraction is closer to real behavior of peer-to-peer networks.

3.2.2 Simulation Logic

Node

Peersim as a simulator of peer-to-peer networks has one major component, the Node. A Node is an abstraction of a computing unit connected in a network, that is a peer-to-peer node.

Nodes in Peersim serve as the master element of the simulator. A peer-to-peer system is characterized by communicating nodes, with a state. Nodes in Peersim respond to messages or events from other Nodes and alter their state accordingly, as well as generate appropriate messages/events for other Nodes.

Peersim organizes Nodes in a network. A peer-to-peer network requires Nodes to connect directly between each other. These connections are called links, the Nodes plus the links between them form the simulated network (see Figure 3.9).

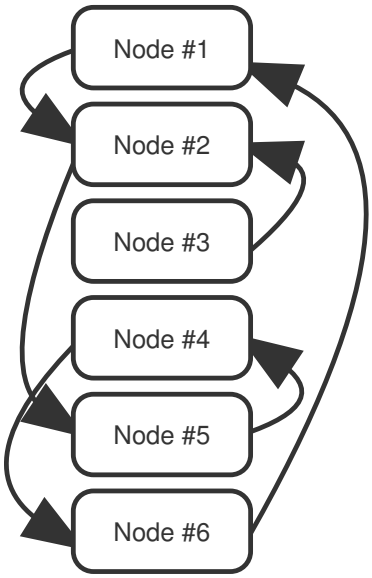


Figure 3.8: A general view of a simulated network

In Peersim how communication is accomplished between Nodes is left to the simulation implementation. The simulator offers an event service to be used by simulations taking advantage of the Event Driven Simulator, however the most common communication process is through shared memory.

Simulation

Over the last sections we have been hinting at the concept of a simulation. It comes naturally that a simulator runs simulations and, in fact this is the purpose of the Peersim simulator.

Simulation A simulation provides a medium for peer-to-peer nodes to communicate between each other, generating observable data in the process.

There are two types of simulation engines in Peersim. In the next sections we will take a deeper look at those engines, still in figure 3.10 we can seen an abstraction of how the simulation is implemented in Peersim, specifically the Event Driven Engine.

A simulation is a very simple loop that offers control to the nodes in the network on an predetermined order allowing the behavior and consequences of the node execution to be observed.

The simulation loop is one of the only two indispensable parts of the Peersim simulator. The simulator has been designed with modularity in mind, simulations run inside one of Peersim's simulation engines, and must implement at least one protocol's execute method. Every other concept described here and from here on is either a guideline for simulation implementers or a shortcut to some behavior that otherwise would have to be implemented from scratch.

A simple simulation consists of implementing the Protocol interface and writing the name of the Protocol class in a configuration file. The executable code of the protocol will be called in a loop and from that point it is the

responsibility of the implementer to define the behavior of the simulation.

At this point it is important to make a distinction that will be valid from here on, when referring to the Peersim simulator and also to DIPS.

When referring to the simulation code we mean the code written by the user of the simulator. This person needs to implement the simulation behavior and there is so little code from Peersim or DIPS involved in the actual simulation that the simulation can be considered entirely composed of user code. On the other hand when referring to the simulator code we refer to the implementation of Peersim or DIPS, this is the code bridges user code and facilitates organization, observation and communication of/in the simulation.

A minimal simulation should implement at least the following components:

- A behavioral protocol
- A data protocol
- A Linkable
- An Initializer
- An observer control

A behavioral protocol contains the main logic of the network, it alters the node's internal state according to the state of its neighbors. The data protocol holds the internal state of the node, the separation between the two protocols, isolates behavior from data so that distinct implementations of each may be tested interchangeably.

A linkable holds the links to nodes defined as neighbors of this node and should be used whenever information must be communicated between nodes. The initializer is closely tied with both the data protocol and the linkable. These protocols hold values and therefore must be initialized, by the Initializer.

Finally the observer control, records data about the progress of the simulation. Might also, optionally monitor the status of the simulation and and indicate to the simulator that it should be terminated.

Protocol

Although nodes are the primary unit of the Peersim simulator, they are used only as a container and usually do not hold any logic.

Each node is associated with one or more protocols that contain the logic for the experiment, *i.e.* protocols implement the algorithms to be tested and are deployed in nodes. A protocol is an object implementing behavior for a node. Protocols exist to isolate independent node behavior, *e.g.* separating communication control algorithms from algorithms to process information.

The node may hold any number of protocols, these protocols may communicate between them within the node as well as with other protocols in other nodes of the network. This way it is possible for a protocol handling communication to receive all communication, pass it to another protocol to be processed, receive the result and send it to another node in the network.

The main advantage of isolating behavior is that protocols may be substituted in each experiment. For instance, it is possible to write several communication protocols, write one protocol to process information and

create one simulation for each communication protocol, in order to understand how each communication protocol compares against each other.

Linkable

As it has been said in the previous section, protocols actually contain the logic while nodes are mere aggregations of protocols. For simplicity we will continue to refer to the nodes as the primary component of the network, *i.e.* referring to them as the executers of an action in the simulation even though a protocol must implement that action for it to be carried out.

Nodes must be able to communicate through the simulator. These components use either the event service or shared memory direct access to execute this communication.

In order to communicate with another simulation component, it is necessary to know who to communicate with. In practice a node must know a set of *IDs* of other nodes known as their neighbors.

It would be possible for the node to iterate through all other nodes in the network and both DIPS and the Peersim simulator do in fact provide a mechanism to accomplish this iteration. In peer-to-peer networks, however, nodes usually only know a subset of the nodes in the network, thus the need for the concept of neighbor, the nodes that a given node knows and is able to directly (without the need to hop through other nodes) communicate with.

Traditionally the hardest part of defining a peer-to-peer network is to design a effective routing algorithm. To help with design isolation, Peersim offers a special type of protocol, a Linkable, which sole responsibility is to maintain a collection of neighbors for a given node.

In order to send a piece of information the node must access one of its linkable protocols and choose one or more neighbors to send the information to.

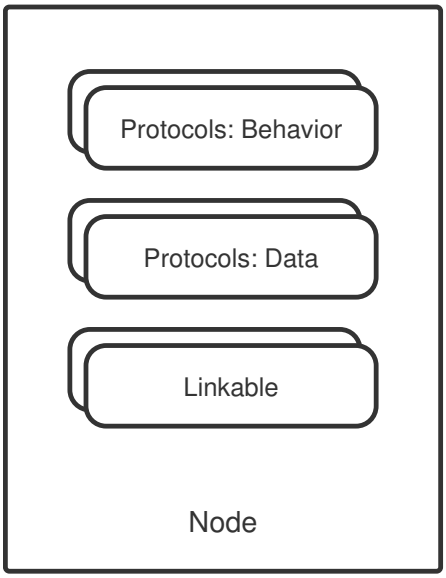


Figure 3.9: Node Architecture

3.2.3 Simulation Control

Time

Time is one of the concepts where DIPS differs from Peersim original implementation. Simulation time is an abstraction that allows control objects to be scheduled, and is an indicator of the simulation progress.

Peersim takes two different approaches to time. In the cycle based simulator, time is the same as cycles, the clock advances by one tick every cycle. This is the approach that resembles the most DIPS. In the event based simulator, Peersim takes a more liberal approach with time. Here, time is an abstract concept that is only present in events.

Peersim time resembles early simulators, in summary events must have a timestamp, when running the event loop, Peersim selects the event with the lowest timestamp, and advances the clock to that timestamp. This allows for events to be scheduled far into the future, and to simulate real network time using statistical delay for message passing.

In a distributed simulator it would be impossible to guarantee delivery on time without damaging performance, and without that guarantee the mechanism is mostly useless, this is why we use a different approach to time in DIPS. Time in DIPS represents the movement of a discrete event sequence. The event sequence is directly tied to the message processing algorithm. Each time a message is processed the internal time on each instance is advanced by one tick. Only simulation events in the main simulator loop advance the local clock, other event such as control messages (in the simulator), execution of control structures (within the simulation) and network events such as connections and disconnections do not have any effect on the local clock. During synchronization states the simulation is paused and the clock is not updated even though control structures may be executed.

There is no synchronization based on global time, the control is done independently of global time and any global ordering of events is not possible.

Control

The last element of a simulation is the Control object. Like Nodes and Protocols, Control objects, or controls for short, are simulation level, user defined structures. Unlike those, controls are not part of the simulated network but live outside the simulation and perform administrative tasks that might alter or observe the simulation.

Controls substitute the network creator running automated tasks to either set variables inside the simulation or retrieve information on the simulation. These objects must be scheduled to be run at predefined times, according to the simulation internal clock. Controls have full access to the simulated network and global state of the simulation.

One of the most common roles of Control objects is to initialize nodes in the network. Controls with this purpose are called initializers.

The simulator initializes nodes with all the protocols needed for their execution, however the initialization of variables inside the protocols is beyond the scope of the simulator.

Separate initialization of protocols is desirable feature has it allows separation between logic, in the protocol, and content, defined by the initializer. It also allows the same protocol to be used in several experiments with different initialization patterns which would, otherwise, need separate protocols. An example of initialization is a protocol that holds a value such as a number or string which different controls would initialize with different values; or a Linkable protocol that needs to be initialize with a set of neighbors.

Another role of Control objects is to generate out-of-network events. These events are not generated inside the network, they might consist in altering the state of a node to mimic an event that would be the consequence of some action taken by an agent external to the network.

An external event might be a simulation of a real world interaction such as user input or a power outage.

External events are extremely important in network simulation. Isolated behavior can often be mathematically proven to be correct, efficient or another metric that might apply to the situation. On the other hand proving these properties when under the influence of extraneous agents is often impossible. Simulation can provide valuable metrics on the behavior of the network under such influence if such events can be mimicked, which they can through the use of a Control object.

The final role of a Control object that we will approach here is the observer role. Controls are the only tool provided by the simulator to take information from the network while the simulation is running. Because they have full access to the network, controls may be scheduled to be run at a predefined intervals and calculate metrics on the network state.

Access to protocol internal variables is often used by controls to calculate the simulation progression and may later be used to compare different simulations or different setups of the same simulation.

3.2.4 Event Simulator

The Peersim simulator supports another type of simulation engine. The Event Driven Simulation engine extends the concept of the simulation making it more realistic than the Cycle Driven engine. In an Event Driven simulation nodes interact with each other through events, an abstract concept that represents outside information for a node to act upon.

The event object is an encapsulation of information that is delivered to a node at a predetermined time.

In Peersim events are scheduled to be delivered to a protocol in a node. Events therefore must contain the information for the protocol, the destination node of the event and the protocol *ID* to receive the event.

The remaining field in a event object is the time at which the event should be processed. As it is stated in the Time section, Peersim supports a model of arbitrary monolithically increasing time model. In an Event Driven Simulation the flow of time is implemented through this field in the event object. The simulation clock is updated to the time the next available event is set to occur.

Events are processed by calling a method on the selected protocol of the node defined in the event. The method receives the information associated with the event, alters the node internal state according to the logic defined for the simulation and possibly schedules events for other nodes in the network.

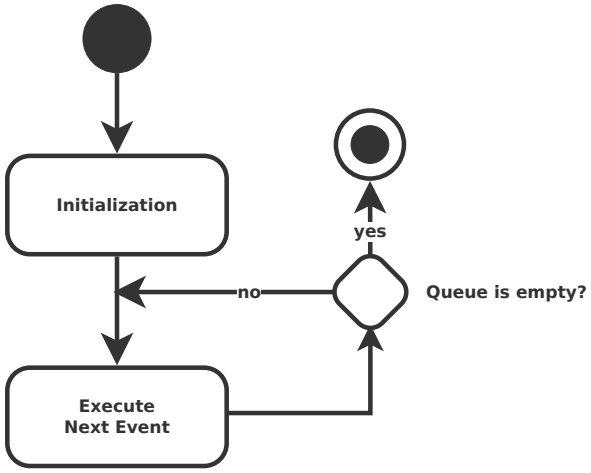


Figure 3.10: Concept of an event based simulation

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.

Initialization

The initialization phase creates the network. The network consist of the structures that allow access to nodes and protocols. 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 network is a globally visible component of the Peersim simulator, this means that simulation level structures have access to it and utilize the provided methods to access 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 expose the entire network.

To summarize, old code and naive implementation 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 initialization components defined in the configuration files. Initialization components are executed exactly as in Peersim, they initialize values and create links. Initialization

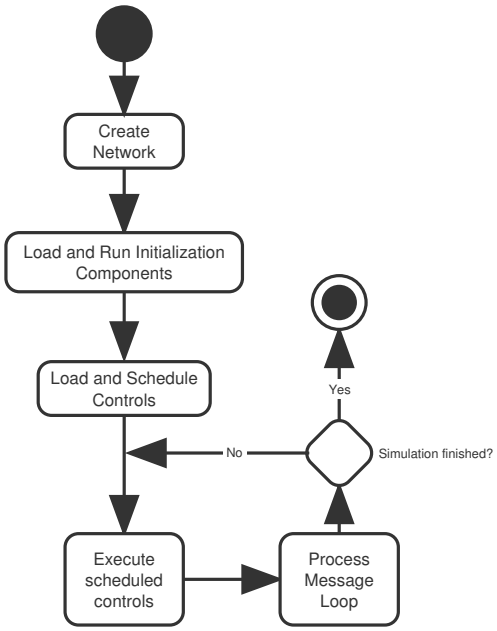


Figure 3.11: Distributed Event Based Simulation - Overview

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.

The final step of the initialization process is to load control components defined in the configuration. In Peersim control components are executed through scheduling, the configuration defines when a control should be executed in relation to the global time variable. In Peersim the global time variable is also 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 is expected to only alter 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.

Event Processing Loop

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

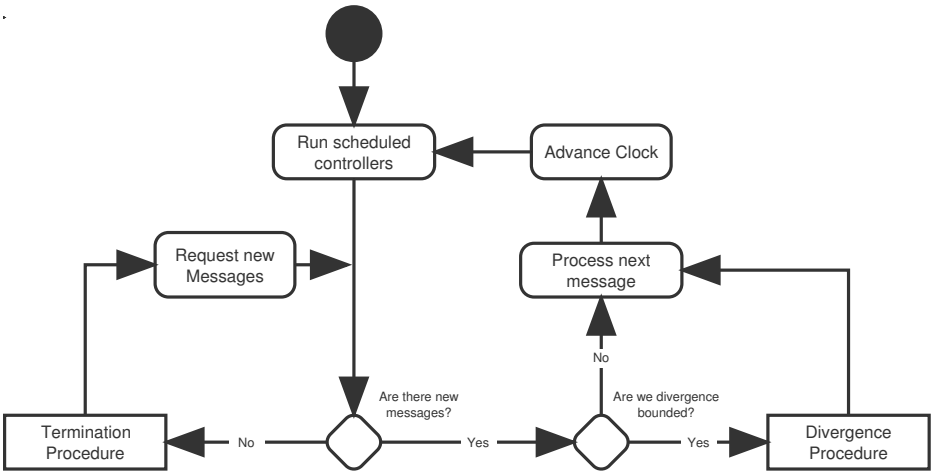


Figure 3.12: Distributed Event Based Simulation - Detail

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 section.

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.

3.2.5 Cycle Based Simulator

Cycle based simulation consists in iterating through all the nodes and allowing each one to execute logic one time for each cycle. Peersim simulations are primarily cycle based.

Unlike event based simulation, cycle based simulations do not have a native asynchronous interface for communication between nodes. Peersim simulations make extensive use of shared memory constructs to simulate interaction between nodes. The distributed nature of DIPS makes it hard to provide shared memory mechanisms.

Any shared memory mechanism, even if asynchronous, would require constant migration of nodes between instances which would create too much of a burden on the simulation speed. DIPS architecture provides cycle based simulation on top of the event simulator.

A node implementation for the cycle based simulator should extend an internal node implementation that

executes the code when receiving a message. An internal cycle control schedules node execution at the beginning of each cycle by sending a messages to each local node.

The cycle control uses the simulation synchronization to guarantee cycle synchronization, *i.e.* each cycle is executed in parallel throughout the network but synchronized at the beginning, every instance must wait for all others before starting a cycle.

Given that in a cycle based simulation there is no simulation level mechanism to handle message passing, we offer a semi transparent shared memory proxy layer. Inter-node communication is possible through this layer. Nodes may access public methods of their neighbors through a proxy that converts this call into messages delivered to the event loop.

There are two types of calls. Methods that do not have a return value use a fire and forget mechanism, once the method is called, the message is sent and execution of the method on the destination node will happen eventually.

Nodes that call methods with a return value must have a mechanism to wait asynchronously until that value is available. DIPS should offer a Future primitive as a return value that would permit waiting for the return values of several asynchronous methods at the same time.

Eventually the code will have wait the availability of the return values. The simplest solution would be to wait asynchronously for the result of the message. This creates large performance drop in the simulation due to the round trip times and would require a separate queue to process asynchronous method calling messages in order to avoid deadlock, this queue would require internal synchronization mechanisms to avoid race conditions as the event loop is centralized. Another drawback is that the called methods would not be able to recursively call methods in other nodes.

The better solution would be to use couroutines. By saving the stack, the program could be continued when the value becomes available, transparently to the user. Unfortunately until the JVM provides native coroutines mechanism, implementations of coroutines in Java and Scala (delimited continuations) focus more on the simpler abstractions such as generators and lack the possibility to resume full stacks. This requires simulation implementations to have full knowledge of the couroutines constructs and use them explicitly in every stack point up to the simulator call.

A solution that sits in the middle of the two proposed above in terms of simplicity and transparency, is to pass high order functions to the wait method of future objects or lists of future objects. This methods would then be called when the values become available. Scala users may pass anonymous functions, while Java users can pass anonymous classes (for instance implementing the functionN interfaces available from the Scala-Lang library). This solution gains in simplicity what it lacks in transparency which we believe is an acceptable trade-off, particularly because a completely transparent solution might hide the asynchronous complexities resulting in deadlocks and race conditions.

Until coroutines are available in the JVM as native construct, complete with full stack switching, we leave the decision on how to implement the asynchronous method calling API to the implementation.

3.3 Advanced Features

The last component of DIPS is actually a loosely coupled group of small components, as it is shown in figure 3.13.

Some of this components serve as a base to others, such as checkpointing, while others act independently, like bounded divergence. What they have in common, is their purpose. All components alter the basic execution of DIPS in order either improve performance or to correct some fault that arises from DIPS distributed characteristic. In this section we will describe each one.

Algorithm 2 Message Bundle

```
###### Bundling of messages
#Pseudo-code on the implementation of the message bundling.
class MessageBundle
    constructor: (size) ->
        this.size = size
        this.messages = new Buffer size

    #Add message to the buffer, return true if the buffer is full.
    add_message: (message) ->
        this.messages.append message
        return this.messages.size >= this.size

(size) ->
    #Join the network.
    dht = new DHT(contact = 'ip:port')

    #Initiate message bundles in instance stubs.
    for instance in dht
        instance.mb = new MessageBundle size

    #Execute while there are messages outstanding.
    while message = receive_message()
        #Find the message destination.
        instance = dht.route message

        #If bundle is full, send messages and create new buffer.
        if instance.mb.add_messsage message
            instance.flush_mb()
            instance.mb = new MessageBundle size
```

3.3.1 Message Bundling

Message bundling is one of the most important advanced features of DIPS. It is a simple mechanism but one that is essential to the performance of the simulator. It consists of buffering the output of the simulator, preventing the

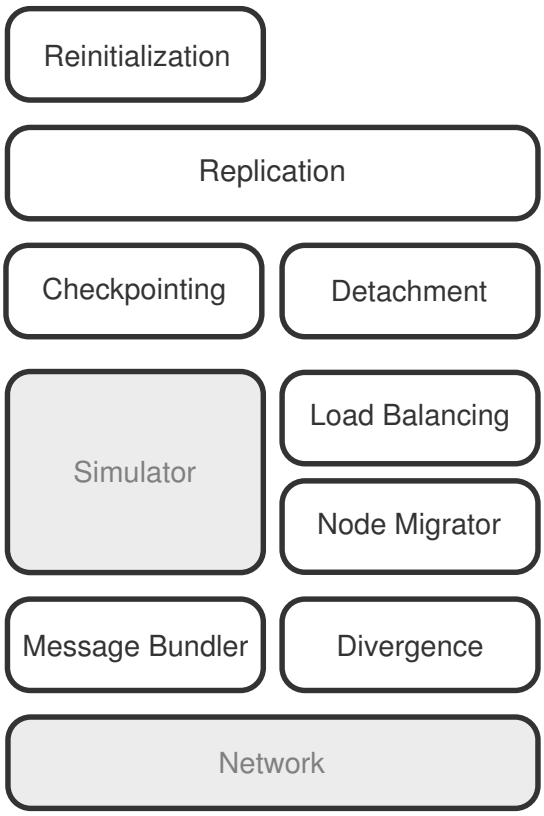


Figure 3.13: Dips architecture detailing the advanced features

congestion that a very large number of outbound messages would create.

Simulation messages must be passed between instances. In a reasonably fast simulator the number of outbound messages can easily reach thousands per second. Sending every message through the network individually places burdens on:

- The communication link — A large number of messages being transferred through the network incur an overhead for each message such as headers, multiple connections, latency for each separate communication.
- The CPU that must process every incoming message — Every message must be serialized and deserialized to be transported, well implemented serializers may be able to optimize (de)serialization if large amounts of messages are transferred together.
- The synchronization algorithms on message queues — Every time messages are transferred from the Post Office to the simulator, thread level synchronization is necessary. This stops the simulation for brief amounts of time that might be bundled together.

Due the number of messages to be transferred between nodes, messages are buffered inside the Post Office before being sent. Messages are sent once a predetermined amount of buffered messages is stored or a predetermined amount of time has passed.

Messages are stored together grouped by destination, once the storing limit for a given destination is reached all messages for that destination are sent in a bundle.

The limits are configurable by the simulation implementer, larger bundles may exacerbate the problem of local clusters defined in the Bounded Divergence algorithm (see section Bounded Divergence), while small limits will lower simulation performance. It is up to the simulation implementer to know how much localized delay the simulation may handle whiteout loosing realism.

Most simulations should be able to handle delay in the distributed simulation transparently, using message bundling together with bounded divergence, *i.e.* delivering messages on request instead of when they are generated.

3.3.2 Bounded Divergence

Soft Bounded Divergence Protocol

Bounded divergence is a mechanism that allows the simulation creator to define the maximum “latency” the simulation can tolerate.

Asynchronous message passing creates unrealistic clusters of simulated nodes whose messages have much lower latency than messages from remote nodes. As locally generated messages are immediately delivered as opposed to remote messages that are bundled together to minimize the strain on the network. This bundling of messages may result in unrealistic latency that we wish to minimize.

Algorithm 3 Bounded Divergence

```
#Example code of a class to handle the message queue
#implementing the **bounded divergence** algorithm.
class MessageHandler
   constructor: (bound_limit, dht) ->
      #The limit of divergence allowed.
      this.bound_limit = bound_limit
      #The network abstraction.
      this.dht = dht
      this.message_queue = new Queue

      #Global object containing the local simulation time.
      current_time = Globals.time

      #Set each instance clock to the current simulation time.
      for instance in dht.instances
         instance.last_seen = current_time

   #Method to be called in order to add a message to the queue.
   add_message: (message) ->
      this.message_queue.enqueue message
      origin = this.dht.route message

      #The *last_seen* field is updated to the current time.
      origin.last_seen = Globals.time

   #Method to be called to get the next message.
   retrieve_message: () ->
      current_time = Globals.time

      #Filters the network list leaving only the ones that have not sent messages
      #for longer than the limit.
      over_the_limit = dht.instances.filter (instance) ->
         difference = current_time - instance.last_seen
         return difference > this.bound_limit

      #Retreive any avilable messages from those instances.
      #This **blocks the simulation** until all responses have arrived.
      for instance in over_the_limit
         instance.get_messages (message) ->
            this.add_message message

      #Return the next message to process.
```

Minimization is guaranteed by processing all outstanding messages within a given time frame. All simulation instances maintain an event counter and and vector counter with one entry per simulation instance including the local one. The event counter must not be more than N events (where N is defined in the simulation configuration) ahead than any of the values in the vector counter. The event counter is updated every time an event is processed. Each vector counter is updated to the current event counter value when, either a message bundle is received from that instance or a request is made to that instance for new messages, which the simulator must do in order to proceed when the event counter reaches the N barrier.

Strictly Bounded Divergence Protocol

The Soft Bounded Divergence Protocol is unable to completely prevent simulation instances to process events at a different rates. It guarantees that all messages generated during a interval of events for nodes localized in a specific instance are delivered within that interval. If any of the instances is either too slow or processing a larger number of events, it is possible that the messages for a given instance are created at a lower rate in this instance than in the others. As a result the messages generated at the slower instance will be consistently **older** that the ones generated elsewhere.

We introduce here a different protocol, instead of simply correcting the Divergence Protocol previously described. We do it for three reasons. It is not necessarily true that a protocol requires all of the simulator instances to run at the same rate, divergence in event processing speed can be a important factor in peer-to-peer protocol testing. It is also worth noting that in order to guarantee similar event processing rates in every instance, the simulator instances must run at the speed of the slowest instance, which is unacceptable in long simulations. Finally the DIPS simulation engine already makes an effort to guarantee load balancing (see Load Balancing) between differently-able instances.

It may, however be necessary to guarantee a synchronization between all instances mainly for protocols that require communication between simulated nodes to **always** occur within a predefined interval.

The Strictly Bounded Divergence Protocol creates event intervals delimited by the bounded divergence parameter, *i.e.* if the bounded divergence parameter is 1000 then interval 0 would be from event 0 to event 999, interval 1 from event 1000 to 1999, etc... To guarantee synchronization between instances an instance may not move to a different interval without contacting all other instances for new messages and must wait until all messages generated locally in the last interval have been processed in the network, *i.e.* to move to interval 3, an instance must have confirmation that all messages it generated during interval 1 have been processed and all messages generated as a result have been delivered.

This will create a deadlock. To solve the deadlock, an instance will process all incoming messages it has procured within the current interval even if it must process more events than the ones defined by the bounded divergence parameter. However in order to guarantee fairness in the processing queue, messages generated by events processed in interval N after the bounded divergence limit will be tagged as $N + 1$.

3.3.3 Node Migration

Algorithm 4 Node Migration

```
class Migrator
        constructor: (dht, local_nodes) ->
                this.dht = dht
                this.local_nodes = local_nodes

        on_network_update: () ->
                tranfer_nodes = this.local_nodes.filter (node) ->
                        dht.route(node) isnt 'local'

                node_migrations = local_nodes.aggregate (node) -> {dht.route node: node}
                msg_migrations = msg_queue.aggregate (msg) -> {dht.route msg: msg}

                migrations = node_migrations.merge msg_migrations,
                             (destination, node_list, msg_list) ->
                                        {destination: (node_list, msg_list)}

                for destination, msg_node_pair of migrations
                        dht.send MIGRATION, msg_node_pair, destination
```

Node migration is the core process that allows the DIPS simulator to dynamically respond to changes in the network. Node Migrator is a module behind load balancing (see section 3.3.4), instance detachment (see section 3.3.6), failure and dynamic network extension.

The organization of the network as a distributed hash table guarantees that changes in the network require only minimal changes to the nodes distribution. This organization requires that nodes may travel through the network from instance to instance. The instance responsible for each node depends on network composition at each instant. Whenever there are changes in the network composition, some nodes will be assign to new instances and, therefore, must be migrated to their new owners.

Node migration requires serialization and transfer of the nodes from their previous owner to their new owner. The non-deterministic nature of the simulation permits that this migration may be executed during *live* simulation, *i.e.* once the a node is assign to a new instance the transfer of that node may be executed in parallel with the simulation, so long as there is a guarantee that the node will arrive at the new instance eventually. Messages destined to the node, arriving still at the old instance, will be forwarded to the new instance. Messages arriving at the new instance before the node is ready, will be delayed until the node is ready.

3.3.4 Load Balancing

As we have seen, nodes in the network may be assigned to simulation instances according to the hash value of their *ID*, using a CHORD like DHT, or using a simple round robin algorithm.

The distribution of nodes throughout the network must be balanced in order to take full advantage of each instance CPU. The round robin algorithm, and the consistent hashing used in the DHT provide an initial version of load balancing. This, given the importance of performance, is not enough. We present here an algorithm to balance the load in an heterogeneous network, that may be composed machines with different hardware and different performance capabilities. This algorithm is based on empirical data gathered live during the simulation.

This distribution may not be fair when only a small number of simulation instances are in use. For example, if only three simulation instances coexist in the network and their hash values happen to be very close in the DHT circle, one instance will be overburdened with most of the nodes, while all others will have very little work.

DIPS has an adaptive algorithm to correct this situation as well as any other situation where an instance is unable to cope with its workload at the speed of the others.

We take advantage of the semantics in the *ID*s of the simulated nodes. Simulated nodes have a numeric *ID* identically distributed between *0* and *N*. The capacity of a simulation instance to handle its assigned work, is defined as a value between 0 and 1, where 0 means it is unable to handle any work, and 1 means that it processes work either at the same rate or faster than any other instance. In order to guarantee an effective load balancing, the DIPS simulation protocol takes this value into account when routing messages. Simulated node responsibility varies during the simulation runtime according to this value.

Instead of simply calculating the hash value of the node, and setting the responsibility to the simulation instance with the closest lower hash value, the protocol requires that a different DHT be created for each node. To create each node's particular DHT, we must filter out all instances that can not be responsible for this node. Only the instances that verify the following formula are considered in the DHT, guaranteeing that overloaded instances become responsible for less and less nodes until they reach an acceptable work rate.

$$NodeID < NumberOfNodes * InstanceWorkCapacity$$

A proposed implementation for the workload indicator for each instance is a measurement of the idle time in each instance. This is possible because a lightly loaded instance will spend a greater amount of time waiting for messages from other instances, while an heavily loaded instance will have to many events to process.

3.3.5 Checkpointing

Checkpointing is the process of saving simulation state. This process is non trivial because it requires synchronization between all instances in the network so that the saved state is coherent. A saved state can serve as a snapshot to be analyzed later and, more importantly, it serves as point where the simulation can be restarted if necessary.

Checkpointing is executed in a synchronized fashion, at each individual instance. It would be better if each instance could perform checkpointing individually and asynchronously however that would require a process to revert changes which would be a burden on memory, greater than full synchronization is on processing speed.

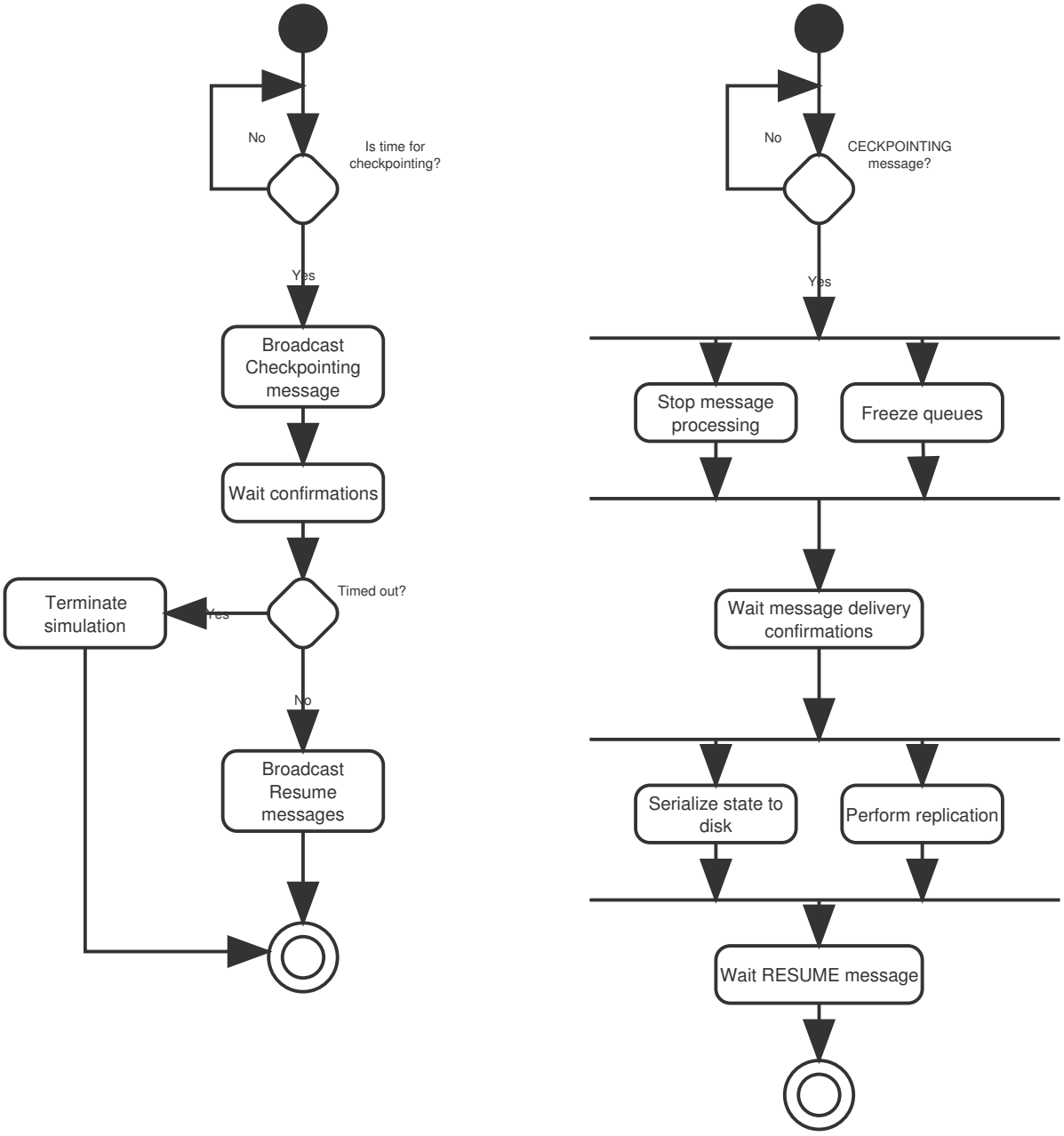


Figure 3.14: The Checkpointing Protocol

In order to guarantee that the process occurs synchronously, the simulation coordinator, as previously defined, solely controls the process. It requires every instance to stop processing, achieve a synchronized state and then perform checkpointing.

The simulation coordinator sets the period of checkpointing as mandated by the configuration options, in relation to its internal clock, corrected of the load balancing factor, *i.e.* the configuration defines that a checkpointing operation should be executed every N ticks. If the load balancing factor for the simulation coordinator has been calculated to be $1/L$, this means the simulation coordinator will initiate the checkpointing operation number X whenever its internal clock is greater than $X * N/L$.

The choice to only account for the simulation coordinator internal clock guarantees that its not necessary to maintain a dedicated line of communication between all instances only to strictly obey to the configuration mandate. The load balancing factor has enough information that we can expect checkpointing to occur within a period close to the ideal clock time, defined in the configuration, as measured by the fastest instance.

The process of checkpointing is started by the configuration simulator when its internal clock overcomes the number previously defined. Initially the simulation coordinator contacts all instances requesting that they initiate the checkpointing procedure, every instance must comply and will acknowledge as soon as the operation has been successfully completed. When all instances have acknowledged, the simulation coordinator contacts every instance allowing for the simulation to be resumed. It is important to note that instances must only resume normal simulation when there is a guarantee that all instances have already performed checkpointing, otherwise their state may become unsynchronized.

From the point of view of a regular instance the process is initiated at the moment when a CHECKPOINTING message is processed (not received, see synchronous control). The instance immediately stops processing messages, incoming and outgoing queues are frozen, which results in incoming messages to be rejected. The instance then awaits acknowledgments on all sent messages. If any of them are rejected, they are placed back in the outgoing queue. At this point, incoming and outgoing message queues, node and controller states are serialized and saved to disk. Depending if, and how, replication is configured replication steps are also performed. The instance then waits for a resume message to arrive.

The modularized nature of the DIPS simulator allows checkpointing to be implemented as a predefined control structure, to be executed at the simulation level rather than the simulator level. It results then, as an extra feature, that a simulation creator might want to run a more time efficient checkpointing protocol, he is free to do so.

3.3.6 Instance Detachment

Simulations may take a long time to process, this is specially true in the case of a DIPS simulation as its size is the main reason to use this simulator. During a very large simulation it may be necessary for the network composition to change, instances may require to be disconnected from the network.

The network component was design with encapsulation in mind, and this is one case where this encapsulation is particularly useful. Detachment may occur during any point in the simulation, it does not require coordination

Algorithm 5 Instance Detachment

```
class Detacher
        detach: (dht) ->
                confirmations = dht.broadcast DISCONNECT
                dht.self_disconnect()
                #migrator.on_network_updates is called
                #wait for broadcast confirmations before exiting
                confirmations.wait()

        on_new_message: (msg) ->
                destination = this.dht.route msg
                this.send msg, destination
```

approval. Other components are only notified that an instance as left the network, because of network isolation, and together with the node migration algorithm, the simulation continues transparently independent of churn.

For an instance to be able to leave the simulation without stopping it, it is necessary to make a few changes to the simulator and network component. First the simulator must handle messages that are not destined to any node present at the local instance. This messages must be re-routed. They might have been destined to a node that was held by the local instance but has been migrated since. They might also be destined to a node that has not yet arrived. The second alteration is that node migration must be triggered as soon as an instance disconnects.

This covers only voluntary disconnection. The protocol requires migration of nodes from the disconnecting instance to other instances in the network, before the disconnection actually occurs. Failure can be handled through another feature, reinitialization.

3.3.7 Replication

Replication is an optional feature of the DIPS simulator. Replication extends checkpointing so that simulations can be restarted from the last checkpoint. Even if the one or more of the instances are no longer available. Replication also permits hot restarts. Hot restarts happen when one of the instances dies before the detachment protocol can be completed. During an hot restart the replicated checkpoint of the now dead instance is unpacked and distributed through out the remaining instances in the network.

As with checkpointing, replication is offered as a simulation level protocol that is offered by the simulator but that can be overwritten in any simulation.

The offered replication implementation simply replicates checkpointing bundles to the closest instances in the DHT. The number of replicas can be configured in the configuration file. The number of replicas can also be considered as the number of instances that can fail within the restart period, in the worst case scenario, without the simulation state being lost.

Although it would be possible to maintain a live replication (*i.e.* keep instance state replicated throughout the

simulation and not only during the checkpointing phase) we considered that this would prove too much of a burden on both performance and memory utilization to be an acceptable option. As a result, replication is only as effective preventing repetition in the face of failure, as the rate at which checkpointing is performed. We have already seen that the synchronous characteristic of the checkpointing process can be quite burdensome to the performance of the simulation, thus also placing limit on the effectiveness of replication. The greatest advantage replication is to guarantee that a lengthy simulation can be resumed from a point **close** to the point of failure, not necessarily to avoid all repetition.

A predefined replication controller is available to all simulations and is initiated when the corresponding configuration property is set. The controller is scheduled to be executed as soon as a checkpointing bundle is created. The replication controller then contacts the next N instances in the DHT, N is defined in the configuration file. Streaming TCP connections are created between the instances, as checkpoint bundles may be too large for the available memory space, the simulator is configured to regularly flush the output of the connection to disk. As soon as the transfer of the bundles is complete the replication process is also complete.

3.3.8 Reinitialization

Reinitialization allows simulations to be interrupted at any moment and later be restarted from the last checkpoint.

Checkpoints are the snapshots of the simulation, taken during a synchronized state. With this snapshots, it is possible to resume the simulation from that synchronized point.

In a distributed system, the largest the pool of resources, the more likely one component will fail. Failure in DIPS is handled rather naively through reinitialization, checkpoints alone take a long time to performed, time during which the simulation must be paused. Other failure prevention methods would require memory overheads that collide with the principal DIPS objective to require the smallest possible memory overhead when compared to Peersim.

Each checkpoint package contains part of the simulation. By restoring each of the packages the network can be recreated as it was at the moment of the checkpoint. Reinitializations are executed at the request of the coordinator. The coordinator will issue this request on one of two occasions:

- After one instance has failed and the simulation cannot proceed, this is called an **Hot Restart**.
- When the user explicitly requests it.

Requests for reinitialization must contain the identification of the checkpoint. As previously stated a checkpoint has an unique identifier defined as concatenation of the hash of the configuration file with a random string generated at the moment of the checkpoint. This identifier must be transmitted whenever a reinitialization request is explicitly generated. It will be selected by the coordinator as the identifier of the last successful checkpoint, in the case of an hot restart.

When performing a reinitialization the coordinator will issue a restart command to each instance, holding the checkpoint identifier. All instances receiving the command will search their paths for checkpoint packages that

match that identifier.

It is not necessary that all instances have a checkpoint package to restore, a simulation might have been stopped and restarted with a larger number of instances. An instance might also have more than one package matching the checkpoint identifier. For this reasons all instances must report back to the coordinator all the package *ID*s found in their path. The coordinator will select one instance for each unique *ID*, and request that those instances restore the selected packages.

At this point the simulation will continue normally. There is however one extra step that all instances that perform restores must take. It is unlikely that all nodes and messages restored at a given instance belong to that instance, actually this would only be possible during an Hot Restart if one of the instances had failed but rejoined the network before the Hot Restart. As some nodes will probably have been restored in the wrong instance, it is necessary for those instances to perform a migration, in order to move nodes and messages to the correct instances.

4 Implementation

4.1 Technologies

4.1.1 JVM

The DIPS prototype was built on top of the existing implementation of the Peersim simulator. This allowed the implementation to focus on the distributed characteristics of DIPS and use part of the already implemented simulation code.

The Java Virtual Machine was chosen as the development environment for the DIPS prototype. The JVM was a natural choice as Peersim is implemented in Java. Nonetheless the JVM offers unmatched cost-benefit relation in terms of performance versus simplicity of implementation.

4.1.2 Scala

Scala is a language that compiles to Java bytecode. The DIPS prototype was developed in Scala as it offers modern language features the current Java specification (Java 7) lacks.

Scala is in its core a functional programming language. Anonymous functions, pattern matching and list comprehensions are first class features of the language.

As the DIPS prototype was built extending Peersim functionality, it is important to understand how interoperability between Java and Scala works. Interoperation between Scala and Java code is seamless, compiled Scala code is indistinguishable to the JVM, from compiled Java code. This means not only Java classes are accessible *as is* to Scala code, but also Scala classes, public methods and variables are accessible to Java code. As it was possible to extend Peersim functionality using Scala, in the future it is possible to extend DIPS functionality using Java.

Although there is no room in this document to explain Scala syntax and language features we will highlight some of it, which will help to understand the next section.

There are some types in Scala that have equivalents in Java which can be more familiar to the reader. `Unit` is the Scala alternative to Java's `Void`. It serves the purpose of representing nothing. The other important type in Scala, for our presentation is `Any`. While there is no equivalent in Java to the `Any` type, the Java's `Object` type equivalent in Scala is `AnyRef`. `Any` is a superclass of both `AnyRef` and the basic types. Finally a `trait` in Scala is the equivalent of an interface in Java, although Scala supports multiple inheritance and does it, by allowing implementors to add code to traits.

It is also important to describe pattern matching to understand the choices made regarding communication. Pattern matching is an extension of a switch case expression. Instead of being limited to equality testing, pattern matching can automatically perform type checking, *unboxing* and casting, all of it in a simple syntax. Because type checking is an expensive operation and pattern matching is a primary feature of the Scala language, Scala has implemented a special construct to create classes optimized for pattern matching, the `case class`. A Scala

Actor is a particular type of class that implements the actor model. The actor is an event loop that processes messages as they arrive. The DIPS prototype implementation uses both *RemoteActor*, actors that can receive messages from remote instances and local *Actor*. The only difference between the two is that the *RemoteActor* must be registered as service, so that the main event loop implemented in the Scala library knows how to deliver incoming messages.

An *Actor* main form of communication is through asynchronous message passing, however the Scala library also provides abstractions to do synchronous and asynchronous request/response calls.

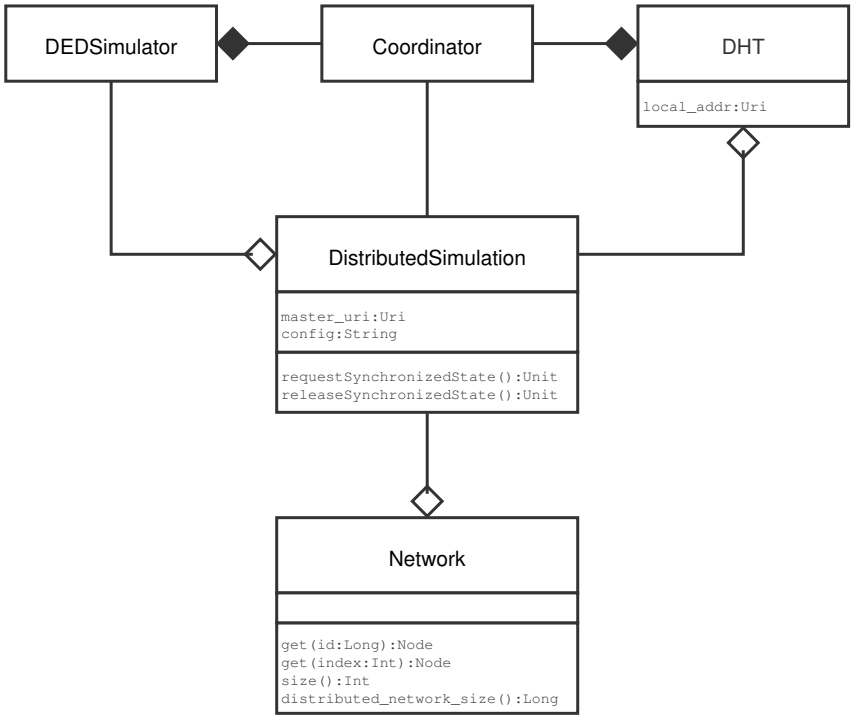


Figure 4.1: Class diagram, a simplified general view of DIPS

4.2 Prototype

The DIPS implementation follows the architecture closely. Where the architecture presented three main aspects, the prototype is composed of three threads of control each one mapping to the respective aspect. The simulator and the DHT map directly to the Simulation and Network aspect of the Architecture. The Coordinator is a controller thread that guarantees all instances act in time and in synchrony, when necessary. The relation between the components is demonstrated in a general, simplified class diagram of DIPS, showed in figure 4.1.

In this section we will go through all three components and try to give an overview of their role. We will also focus on protocols and behaviors that have derived from the implementation and that are not present in the architecture.

4.2.1 DHT

The general diagram of DIPS in figure 4.1 show that the DHT is present in almost all steps of DIPS execution. It is created even before the Coordinator and is started by the Coordinator. From this point on, all components that need network communication go through the DHT class.

The DHT class is the front end, responsible for network communication. It offers three main methods:

```
send(Routable)
sendTo(Any, Uri)
broadcast(Any)
```

The send method takes a message, from which it can extract an address through the routing method, and calls sendTo. An Uri is a tuple storing the address of a remote instance, an *ip* and a *port*. *broadcast* and *sendTo* are self explanatory.

Most of the code in DHT is specific to DIPS and some of it breaks the isolation that is defined in the architecture, for this reason general network code was pushed to a superclass, *PostOffice*.

The *PostOffice* class extends the the native *Actor* class in Scala and is the primary receiver of network communications. It was important to separate this from the network, the *DHT* class extends the *PostOffice* class.

PostOffice only understands two types of messages, *RoutingEvents* and anything else. Both of them are delivered to the *DHT*. The timing to deliver this messages is different, the routing events are delivered immediately on arrival and the processing code is run on the *PostOffice* thread (the PostOffice is an Actor therefore runs on its own thread). The other messages however are stored in a buffer and are only delivered on request.

The instance class is a proxy to a remote instance in the network. It holds information on the remote instance, such as the *ip* and *port*. It also maintains an actor to handle communication offering one method:

```
!(Any)
```

The *!* is the standard method name for sending messages to actors in Scala. This method sends the message asynchronously guaranteeing that it will eventually arrive.

Services

Broadcasting in the prototype is achieved by sending the message to all instances in the network. Given the small number of instances this is not too much of a burden on network performance.

Broadcasting could be implemented by cycling through the network, using flooding or other type of algorithm, it was our view however that broadcasting is not scalable therefore it is better to take advantages of this mechanism when the number of instances is small. At the same time, use as little as possible so that scalability can be achieved through other methods when it is needed.

The network implements a simple publish/subscribe protocol to ease the communication with external modules that require arriving messages to be immediately delivered.

There are two special type of classes to use in the publish/subscribe protocol:

```
Publication(name:Symbol, msg:Any)
Subscription(name:Symbol, msg:Any, sender:OutputChannel)
```

Additionally there is a companion method in the network class that allows an actor to subscribe to publications with a given name:

```
subscribe(name:Symbol, actor:AbstractActor)
```

This protocol is currently used by the coordinator actor, presented in section 4.2.2.

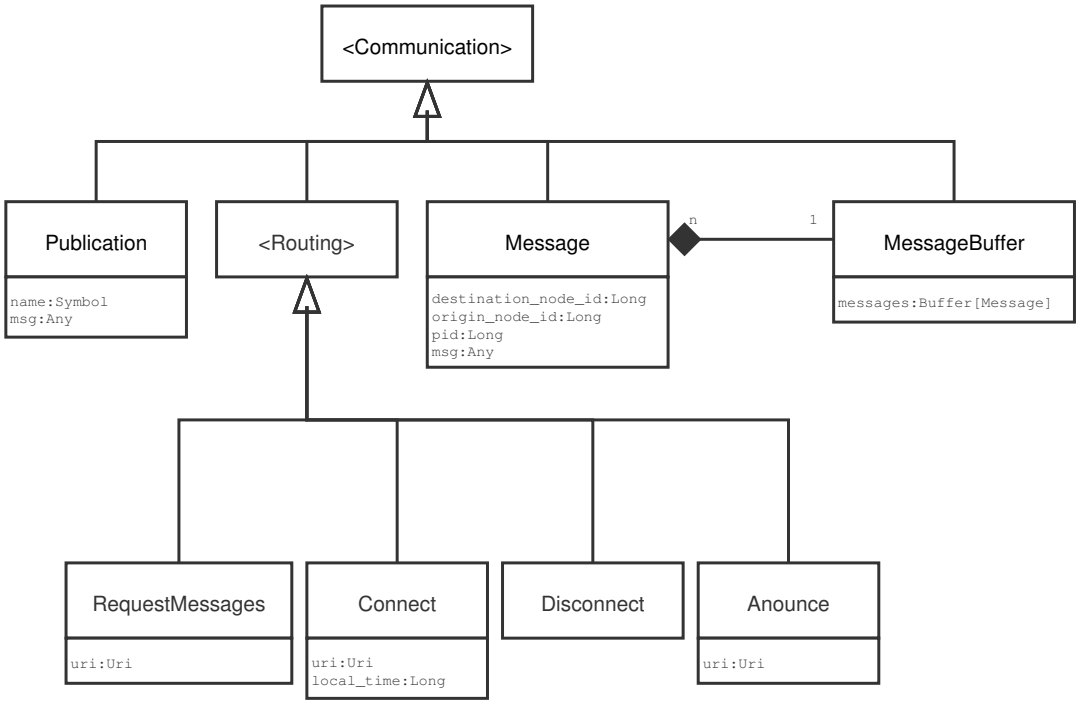


Figure 4.2: Diagram of classes used in communication

Communication Types

Routing events, types implementing the trait *Routing*, are types dedicated to the management of the network. The class diagram that shows the relation between these type is shown in figure 4.1. These types carry information regarding network actions. The available routing events are:

```
case class Connect(uri:Uri)
case object Disconnect
case class Announce(uri:Uri)
case class RequestMessages(uri:Uri)
```

Connect and *Disconnect* serve the purpose of respectively connecting and disconnecting from network. *Announce* carries information about a newly connected instance to the network, information that is sent to other instances so that they can update their routing tables.

RequestMessages, requests messages from other instances when the simulation is idle. From the architecture, it should be outside the network, as the network should not be able to understand what a message is. The simulation message queue was, however, moved inside the network for performance reasons, therefore this request must be handled at the network level.

The *Message* class is a particular type that is bounded to the simulator, a message is information sent from one simulated one to another simulated node. In the simulator section we will see how a message is composed, to the DHT a *Message* is just a subclass of *Routable*, and as such it can be routed to a particular destination.

Extended features

Although the architecture has the message bundling organized under the advanced features aspect, in terms of implementation it becomes simpler to directly place the message bundling features inside the instance class. This way it is possible to extend the instance class overloading the *!* method. This way it is possible to intercept sent messages and place them in a buffer until send conditions are met. As a result all the messages are sent together.

Although it is not part of the DIPS architecture to offer any type of clock synchrony, for testing purposes we decided to implement a rudimentary protocol. This is a very simple protocol that tries to reduce the clock delay between heterogeneous hardware, early testing indicates that clock error stabilizes at about 100ms after synchronization.

This is a maintenance feature, as such it is important that it does not affect DIPS performance with extra control messages. The synchronization protocol consists of only two control messages piggy-backed on the connection control messages. A connecting instance asks the instance it is connecting to, what is the current network clock is, it receives the value back tries to approximate the round trip time and sets its own network clock to the resulting value. The formal algorithm is described in Algorithm 6.

Algorithm 6 Clock Synchronization

Connecting Instance:

$intialTime \leftarrow getCurrentTime()$

$delay \leftarrow sendConnect(initialTime)$ {receive delay as response}

$rtt \leftarrow getCurrentTime() - initialTime$

$setDelay(delay - rtt/2)$

Reciving Instance:

$delayedTime \leftarrow receiveConnect()$

$delay \leftarrow getCurrentTime() - delayedTime$

$reply(delay)$

4.2.2 Coordinator

The Coordinator is a class that, like the DHT, extends the *Actor* class of the Scala library. As we can see in figure 4.1 it plays a central role in the organization of the instance. The coordinator is initiated when the prototype starts, coordinating both the simulation and the network from there on.

The Coordinator starts the DHT, and contains the logic to coordinate instances. Contrary to what the architecture may lead to believe, particularly in the advanced features section, most of the advanced features of DIPS require either strong coordination or actual synchrony.

In this section we present some of the coordination protocols we implemented in the prototype. The token negotiation permits assigning a master to the network, capable of taking centralized decisions. Synchronized state is the protocol to pause a simulation in a state that is synchronous, which is necessary to create checkpoints. Termination is a way to terminate a simulation when there are no more events to process.

Termination

Simulation must stop at some point, this is simple in Peersim because the simulation may stop as soon as the event queue is empty. Achieving a termination state in the DIPS prototype is a non trivial task due to its distributed nature

A simple termination protocol could be described as:

- Instances inform the master of their willingness to terminate.
- Master sends termination command.

Unfortunately, data may be lost in the wire. It is possible for instance A to be empty, inform the coordinator, than instance B becomes empty, and also informs the coordinator. The coordinator issues the terminate command, however in the meantime instance A has received data that was sent by instance B before it had became empty. A correct algorithm must abort the procedure when this happens.

An intuitive solution is that whenever an instance that was previously empty receives new data it also informs the coordinator. This solution still suffers from desynchronization, the coordinator may still issue the terminate command before receiving the update on the instance's status.

The protocol implemented by the DIPS prototype is based on the three-phase commit protocol. We add an extra step, before issuing the terminate command, the coordinator issues a prepare to terminate command, upon receiving this command, the instance must stop accepting messages, making sure all outstanding messages remain at the origin, and guaranteeing no message is lost in the wire. The coordinator must then wait for a positive acknowledge from all instances in order to terminate, if any instance replies negatively, implying that it has outstanding messages to either process or deliver, then the coordinator sends an abort command to all other instances and simulation continues normally.

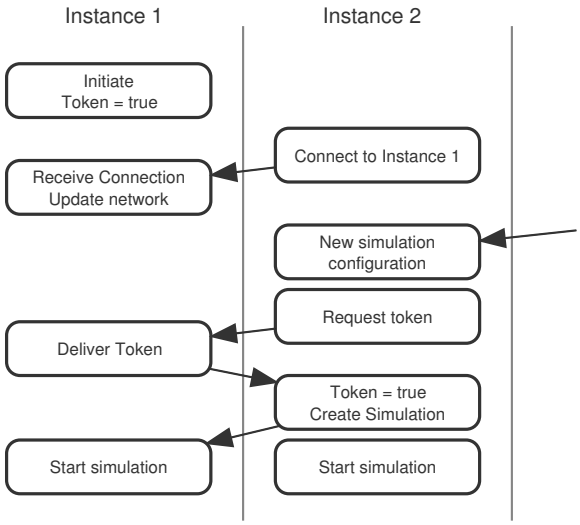


Figure 4.3: Token Negotiation when starting a simulation

Token Negotiation

Token negotiation was briefly referred to in the architecture. In a distributed network there are always decisions that must be centralized either because it is not possible to take them in a distributed manner or because it would be too expensive to implement a distributed protocol for a simple use case. This is why we have left open the possibility for the network to be controlled at a given point by one central instance.

In the prototype, this centralized behavior is necessary on three occasions. One of them presents it self immediately. Because the simulator is composed of several instances, it must create the network before starting the simulation. When and which simulation to start is a decision that should be centralized. The prototype does this taking advantage of the token protocol, in figure 4.3 there is a representation of how two instances negotiate the token, when one of them wants to start a simulation but does not hold the token.

The other occasions when the token protocol is necessary happen during the execution of the simulation. Only the master instance may pause a simulation to achieve a synchronized state. And finally, when a new instance enters a running simulation it is the master instance that instructs the new instance to initialize internal state to join the simulation.

4.2.3 Simulator

The simulator is a class responsible for the simulation. The actual class name is *DEDSimulator*, which stands for Distributed Event Driven Simulator. We called this section simulator because in the prototype the *DEDSimulator* is the only available simulator.

Although the architecture calls for two simulators, the implemented one and the cycle based simulator, we considered that the event driven one was enough to prove the correctness of DIPS, and given the limited time available for the implementation it was preferable to implement more advanced features than, duplicate work,

creating another simulation engine.

Distributed Event Based Simulator

Figure 4.4 shows that the *DEDSimulator* contains a list of controls and a network, which is a collection of local nodes, and has access to the DHT. It would be expected that the simulator would also hold a message queue, however the message queue was moved inside the DHT for performance reasons, this way incoming messages can be placed directly in the queue on arrival instead of being buffered and later moved to the queue.

The simulator implementation follow the architecture very close, there are only two main differences that come from the distributed nature of the simulation, from which it was not possible to insulate the simulator.

The first particular case in the simulator implementation, is how to handle pausing. The simulator uses signals and the *DistributedSimulation* class to communicate with the coordinator. When the coordinator places the simulation in a paused state, the simulator will be blocked at the initial stage of the event loop. When the simulation is resumed the coordinator signals the simulator to continue.

During the paused simulation phase, the simulator is not completely paused. Controls may use this time to perform maintenance tasks, so while the simulation is paused the simulator maintains a loop running event driven controls, *i.e.* controls that receive messages from other controls.

The other case happens when the queue runs out of messages. The simulator, before blocking while waiting for new messages, asks the DHT to send out *RequestMessages* messages in hope that there might be some blocked messages in the network.

Node, Protocol and Linkable

Node implementation in the prototype is mostly the same implementation as in Peersim. In fact the simulations implemented to test the prototype, presented in section 5.1, use the *GeneralNode* implementation provided by the Peersim runtime. The only difference between DIPS nodes and Peersim nodes is that DIPS nodes must be serializable, which is a requirement of node migration.

In the case of the protocols implementing the *Protocol* interface, we needed to change the implementation in relation to Peersim. The reason for this change is the same for protocols, for linkables and for the network. Peersim makes wide use of pass-by-reference variables, this is the case when generating a message for any given node.

DIPS is distributed and only a portion of the nodes are present in the local instance so passing the reference to the actual node would not be satisfiable. Although we could have created a proxy to make it appear that a node was actually present, for performance and memory reasons, we decided to change the signature of the *send_message* method in the simulator as well as the *process_event* method in the protocol, these now must reference the destination node by its ID which is a unique *Long* value.

This is the reason for the most of the incompatibility between DIPS and Peersim simulations. While most controls run in both, protocols and linkables must be altered to substitute direct node references with references

by *ID*.

Distributed Network

The network suffers from a similar problem as presented in the previous message. It provides access to nodes in the simulation by either *ID* or index. With access by *ID* there is no problem, either a node is local and the network returns the node, or the node is remote and an exception is triggered. A component written for a remote simulation should be able to handle this use case.

Unfortunately the primary access to the nodes in Peersim is done using get by index. There is no way of knowing what is the index of local or remote node. While the DHT can be queried with the ID of a node to know whether the node is local or remote this is not possible with indexes.

Our solution was diametrically different than the one used in the previous section. We decided there were no grave losses in allowing get-by-index, and that there was some gain in allowing blind controls, *i.e.* controls that do not know they are being run in a distributed network.

Components accessing the network get-by-index method are presented with a pseudo-index that is only available for local nodes. Blind components only see the *Network* interface, components built to be distributed may access distributed methods of the Network class, using the *DistributedNetwork* interface.

Distributed Configuration

Simulation configuration is created through a configuration file with directives. These directives range from, which class to use as a node, what protocols and controls to load, initialization values for those protocols, and others. Further information on the syntax of the configuration file is available in Peersim documentation and Peersim source code.

Distributed configuration maintains the same principles of the original Peersim configuration. The configuration file is sent through the network to one of the running instances which will then start a simulation with that configuration. The shared configuration file will be responsible for the configuration in each instance as, it would on a single instance simulation.

Only two configuration directives have been created specifically to the distributed simulator. A configuration file must have the distributed keyword in order to select a distributed simulation (the DIPS simulator is able to execute non distributed simulations). The configuration file must also have a *distributed.name* directive whenever checkpointing is used, to distinguish between different simulations, and also, to give a meaningful filename to the checkpoint file.

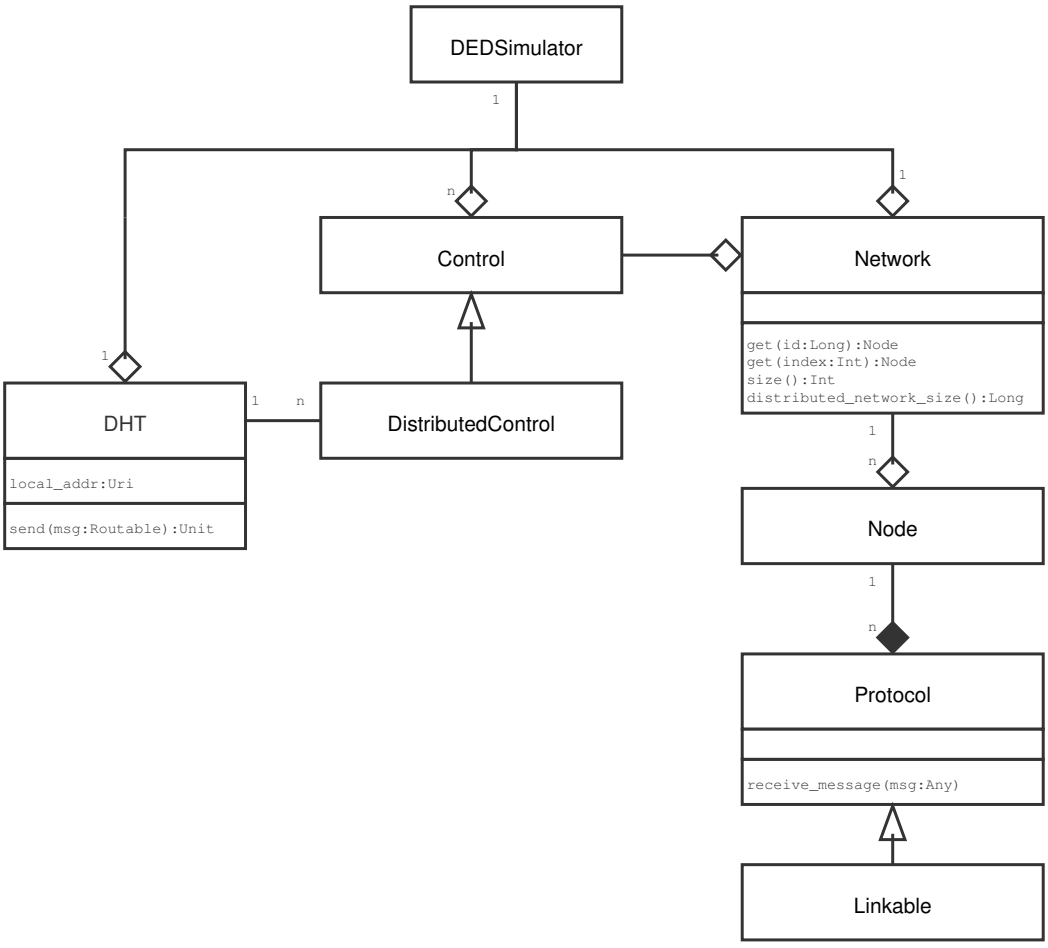


Figure 4.4: Simulator Class Diagram, detail classes involved in the local simulation

5 Evaluation

5.1 Simulations

The evaluation of the DIPS simulator requires the creation of simulation definitions. DIPS usage statistics will be collected from the simulations execution.

Even though we tried to achieve maximum compatibility between simulations created for Peersim and simulations for DIPS, the prototype implementation requires the following changes to the simulation definition:

- Node must be identified by ID instead of direct memory reference, as it they are in the Peersim API.
- In DIPS, the *process_event* method of the *DistributedProtocol* interface has a different signature from the *process_event* method in the *EDProtocol* defined by the Peersim API.

We have built two specific simulations for the DIPS prototype to test DIPS features. These simulations, presented in the next two sections, will be used throughout the tests in order to collect information on the behavior of the DIPS prototype.

5.1.1 Infection Simulation

Infection is a protocol mimicking a viral infection. From a technical perspective Infection is a simple predictable protocol, it is a forward only protocol, *i.e.* nodes never respond to messages they receive, their reaction is always forward looking.

This is not a very good protocol to test compliance, as the protocol does not break or get into an incoherent status caused by late or dropped messages. But from a performance, resource usage perspective, its predictability and tolerance to byzantine behavior makes it a prime candidate to these types of tests. It will serve as the base for comparisons on most of them.

Infection works using the following algorithm:

1. A node sends an infection message to a random neighbor.
2. A node that receives an infection message will increase its infection count by one.
3. If the infection count is higher than the limit, the node is dead, it will not send any more messages.
4. If it is lower than the limit it will select a random neighbor and send an infection message to it.

Infection will eventually end when a percentage of nodes is dead, *i.e.* when the last remaining message was delivered to a dead node. One of its most interesting aspect is that the number of messages sent is predictable. The configuration allows to specify how many messages a living node should send out when it receives an Infection message, we call this the *degree* of the simulation. The default configuration is 1 to 1 but it can be changed to any value (1 to 2, 1 to 3, 1 to $\frac{1}{2}$).

5.1.2 Average Simulation

The Average simulation is a simulation adapted from the Peersim example simulation. This simulation starts by randomly setting a value in every node in the network, then the simulation continues by getting a node to average its value with one of its neighbors.

Like Infection, Average also implements a very simple protocol, but the results are diametrically different. The Average simulation requires nodes to answer messages received, with their old value so that the sender node may also set its value to the average of both. By using this asynchronous communication protocol, the simulation becomes very sensitive to message latency.

The Average simulation converges. At a given point in time all nodes will have the same values, as long as the network is closed.

It is possible to follow the evolution of the simulation by periodically calculating the variance of the nodes values. This metric should ideally be monotonically descendant until it reaches zero. When the variance does not converge, this is an indication that the messages are being delivered with a significant difference in relation to their creation time total ordering.

5.2 Hardware Specifications

The evaluation was preformed using virtualized hardware provided by the Amazon Web Services (AWS) Elastic Compute Cloud (EC2). AWS EC2 offers a cloud based infrastructure with very quick deployment times, virtually unbounded scaling and well defined hardware specifications.

AWS EC2 offers several types of instances with varying hardware specifications. The instances used for the evaluation of the DIPS prototype where:

- **Small instance** — 32bit instance with 1.7GB of memory and a single core with 1 EC2 Computing Unit (equivalent CPU capacity of a 1.0–1.2 GHz 2007 Opteron).
- **Medium instance** — 32bit instance with 1.7GB of memory and dual core, each core with 2.5 EC2 Computing Units.
- **Large instance** — 64bit instance with 7.5GB of memory and dual core, each core with 2 EC2 Computing Units.

Tests described in this chapter were run on Ubuntu Server 11.04 running the OpenJDK Java Runtime Environment.

5.3 DIPS Simulation Compliance

In this section we will demonstrate the adequacy of the DIPS prototype to comply with the semantics and the functionality originally provided by Peersim. The first goal of DIPS is to mimic Peersim behavior, and offer a

converging simulation. No total ordering of messages is provided by DIPS, as no peer-to-peer simulation should require such total ordering. However, a simulation may expect a reasonable latency in message delivery, which we will test in this section.

Three types of tests were performed.

- Compare simulation results between DIPS and Peersim.
- Test whether network latency and message bundling generate artificial clusters in local machines.
- Test whether messages generated remotely have excessively higher latency than messages generated locally.

5.3.1 DIPS and Peersim Convergence

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. Admitting that Peersim is an adequate peer-to-peer simulator.

In Figure 5.1(a) we plotted the variance of an Average simulation of 10000 nodes where the nodes were initialized with linear values from 0 to 100. Next to it is a nearly identical graph in Figure 5.1(b) that represents the same simulation when executed by the DIPS simulator on one instance.

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.

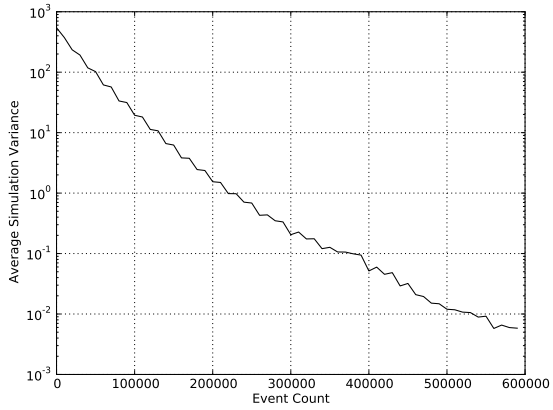
If simulations converge on DIPS running on one instance, the obvious question is: what does it take for them to converge when run on more than one instance. By analyzing the architecture of DIPS it becomes apparent that the message queue on each instance is key to whether or not a simulation will converge. 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.

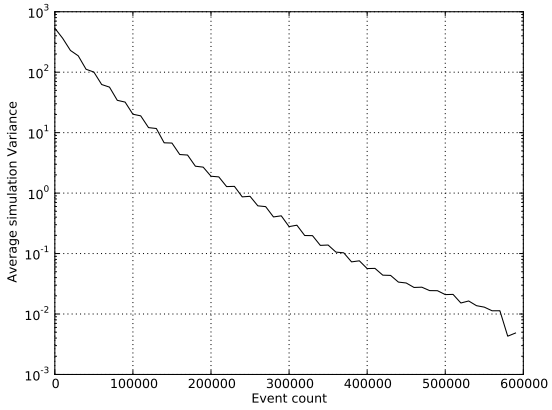
5.3.2 Artificial Local Clusters

We must test the DIPS prototype in relation to the creation of local clusters. Intuitively, because messages to local 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



(a) Peersim



(b) DIPS

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. As stated in section 5.1.1 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 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 5.1 and 5.2 . Figure 5.1 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 demonstrate 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 5.2 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.

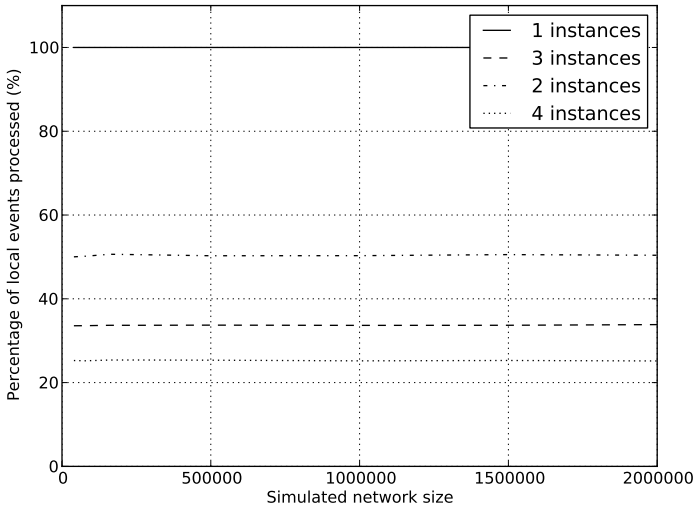


Figure 5.1: Infection: percentage of local events with degree=1

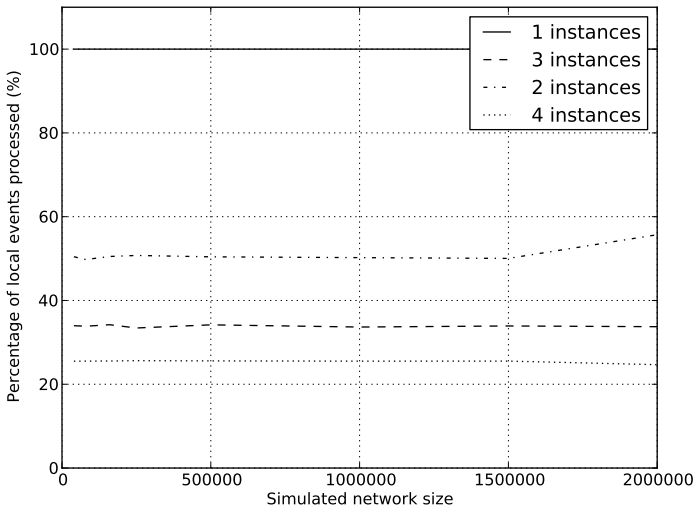


Figure 5.2: Infection: percentage of local events with degree=3

This test shows that all the remote messages generated, will eventually be processed. It says nothing of when those 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.

5.3.3 Local to Remote Message Deviation in Latency

The last test performed to show the adequacy of the DIPS simulation, tests local to 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 500000 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 the clock synchronization algorithm presented in section 4.2.1, to be able to compare average message delay between remote and local messages. Tests on the same machine indicate that the protocol error is close to 100ms. With each message delay calculated, the average message delay for the instance was updated. Separate average message delays 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.

Results show that some of the variables in the simulation are orthogonal to the average message delay. Instance count and network size do not 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 5.3 we plotted the average message delay as a 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,

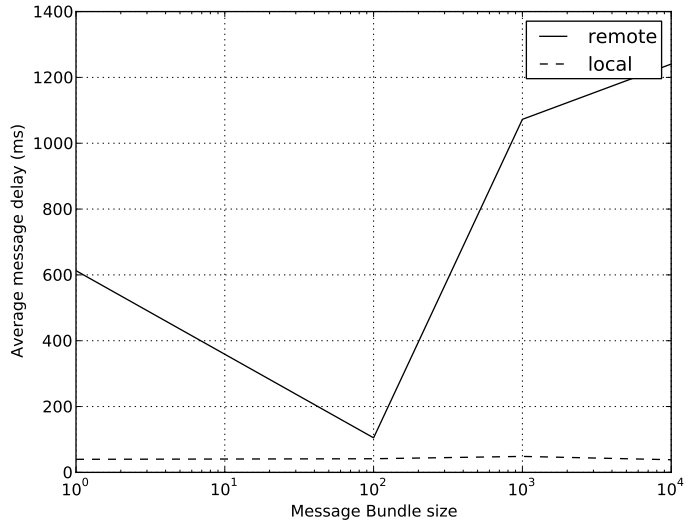


Figure 5.3: Infection: comparison of local and remote average message delay in simulation with degree of 1

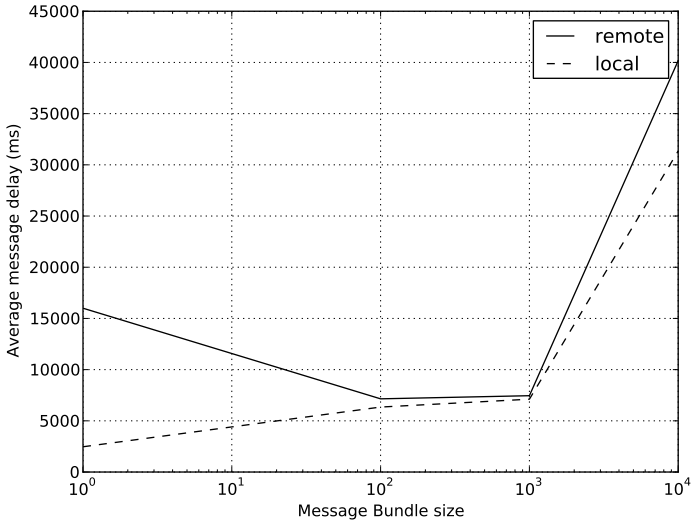


Figure 5.4: Infection: comparison of local and remote average message delay in simulation with degree of 3

under the clock error.

In figure 5.4 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.

5.4 DIPS Performance

In this section we will test performance of the DIPS prototype. Our second goal regarding DIPS, was to create a simulator that would eventually be faster than Peersim. DIPS must then, be monotonically faster with every added instance to the network.

5.4.1 Comparing DIPS to Peersim Event Processing

In this test we evaluate the event processing speed of DIPS in relation to Peersim. The purpose of this test is to compare the performance of an Average simulation running on the Peersim Event Driven engine and, on the DIPS Distributed Event Driven engine using a single instance.

We include this test, as a a distributed implementation of Peersim must be tested against Peersim. It is important to note, however, that it is not a goal of DIPS to be faster than Peersim. The goal stated in the beginning of this document is that a simulation in DIPS should eventually run faster than in Peersim. This could be proved simply by proving that adding instances to a DIPS simulation makes DIPS monotonically faster.

DIPS running on a single instance is architecturally identical to the Peersim Event Driven Engine, any gains or losses in speed depend only on the implementation, and how optimized it is. This test is, nonetheless, an important tool to normalize results of the other tests in this section.

We have consistently preferred the Infection simulation to run the tests. The reason behind this preference is that the Infection simulation is more configurable than the Average simulation, for instance it has allowed us to test situations of starvation and over production of messages, that the Average simulation can not. In the case of this test we chose the Average simulation because, we feel it would be a more correct comparison. While Infection was built for DIPS, and could be adapted to Peersim, Average was built for Peersim and has been adapted to run in DIPS, which is closer to the spirit of DIPS: run Peersim simulations, distributed.

We run the Average simulation with the network sizes of 40000, 80000, 160000, 250000, 500000 and 1000000, 1500000, 2000000. The simulation was run for 600000 events after which it was stopped. In the end of the simulation, the number of events processed was recorded as well as the time passed. This experiment was run on an AWS EC2 medium instance.

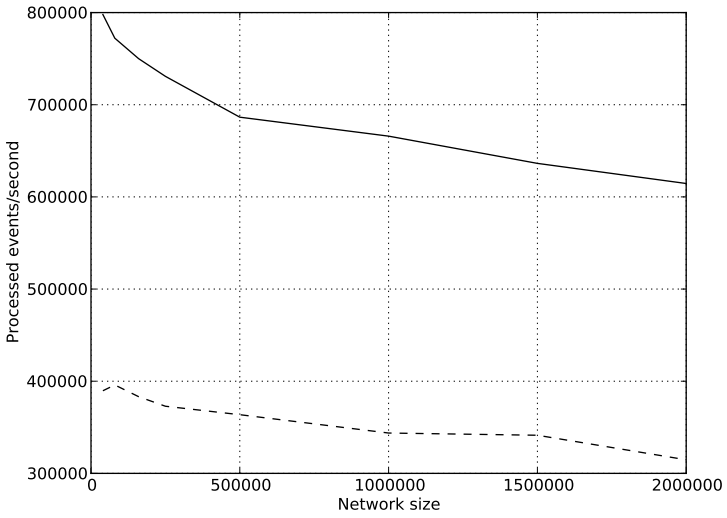


Figure 5.5: Peersim and DIPS running on 1 instance performance comparison

Although both simulators are architecturally identical, DIPS prototype implementation uses some structures that should make it consistently slower, *e.g.* the DIPS prototype uses a hash map to store nodes, while Peersim uses a simple array, making lookup slower in DIPS.

The results presented in Figure 5.5 show that, in fact the DIPS simulator seems to be slower than Peersim by a factor of two. We interpret this results as the consequence of two particular structures unique to the DIPS simulator. First, the structure of the network table that is an hash table in the case of DIPS but a simple array in Peersim. Second, even when running on a single instance the DIPS simulator performs routing on every node, even though the only instance in the network is the instance performing the routing.

These results were expected, even if underwhelming. It should be possible to optimize implementation so that DIPS runs at a closer speed to Peersim. It is important to node that this are only implementation differences, it does not take into account any of the network control and synchronization which are not relevant when there is only one instance in the simulation.

These results also showed another face of the simulation that seemed contradictory. When looking attentively to Figure 5.5 we can see that the processing speed at both Peersim and DIPS is only slightly influenced by the network size. This behavior is intuitive if we approach it from the high level view of the event driven simulator presented in the architecture. It also contradicts all other tests we had previously run in either DIPS or Peersim.

After some search we found the contradiction. It is extremely common for a simulation to have an Observer control that is executed every N events, in fact it is so common that all our tests before this one did. Usually the job of the Observer is to iterate over the whole simulated network and compute some statistic. One of our tests, for instance, indicated that the simulation would make $9 * 10^5$ routing requests within a period of $6 * 10^5$ events while in the same period a single control would make $3 * 10^7$ requests.

This tests leaves us with a slightly paradoxical conclusion: it is true that in both Peersim and DIPS, simulation

processing speed is almost independent from simulated network size, however, almost no simulation produces any results without an iterative control, and the execution of such a control makes the simulation processing speed linearly dependent on the simulated network size.

5.4.2 DIPS Event Processing Speed

In this test we will compare how several instances of DIPS running the same protocol, respond in terms of processing speed.

We selected the Infection simulation to run this tests so that different degrees could be selected, showing us the differences from a simulation in the process of starvation and a simulation producing more messages than it can consume.

Processing speed as we saw in the previous test is a complicated metric as it changes profile, from constant to linear, with changes in the simulation configuration. To run this test we had to decide whether to schedule an Observer control as it would probably happen in a real simulation, whether to schedule this Observer control with a carefully selected schedule so that the impact of the Observer control in the simulation would be minimized or simply to leave it out has we had done with the previous test.

From scientific perspective the third choice would appear to be the best, as it isolates the behavior of the simulation which is what we are testing. We must, however, take a deeper look to make a rational decision.

As we have said before, almost no simulation can create meaningful content unless it runs with an Observer control that can extract information from the running simulation. This extraction usually requires the control to iterate over the simulated nodes. Most of the times the iteration is actually more computationally expensive than the simulation, for sensible configuration values. Such as running the Average protocol with an Observer protocol running every 10000 events.

So we have an operation that is extremely important to a successful simulation, that is frequently more expensive than simulation it self, and finally that has parallelization characteristics that could benefit from a concurrent distributed simulation.

We decided to include the control in the simulation as it would not be correct to announce the, possibly, speedup of Peersim using DIPS that could only be reproduced under unlikely configuration with slim to no chance of actual reproduction under real world environments.

The simulation runs a predetermined number of events which was set to 500000. During the simulation a control would save the number of processed events per second up to that point in time. This control would run every 25000 events. After the first instance reached 500000 events the simulation was stopped and the data from each instance would be collected.

The simulation was run on AWS EC2 on medium instances. With message bundle of 1000 and using combinations of:

- Instances: 1, 2, 3, 4
- Network size: 40000, 80000, 160000, 250000, 500000, 1000000, 1500000, 2000000

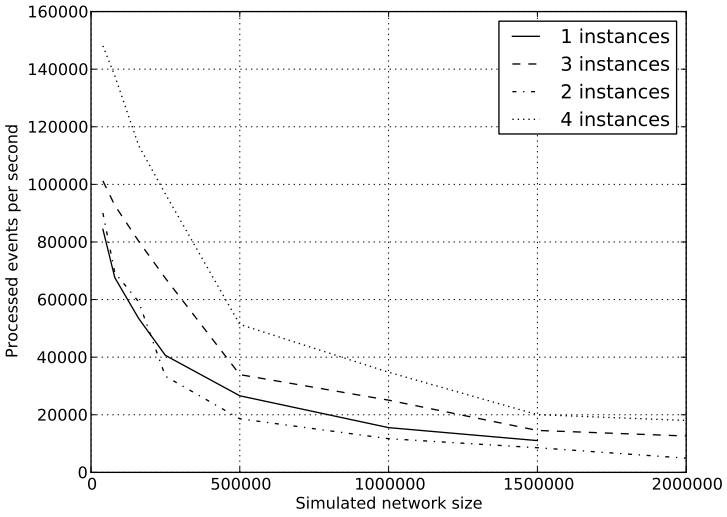


Figure 5.6: Simulation processing speed with degree=1

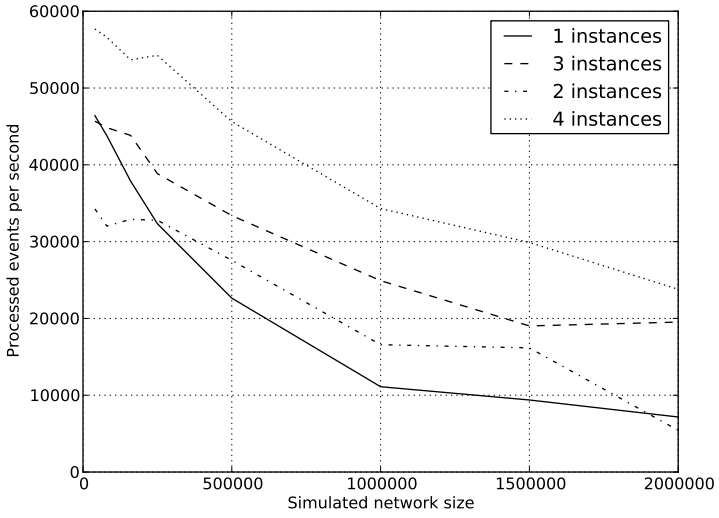


Figure 5.7: Simulation processing speed with degree=3

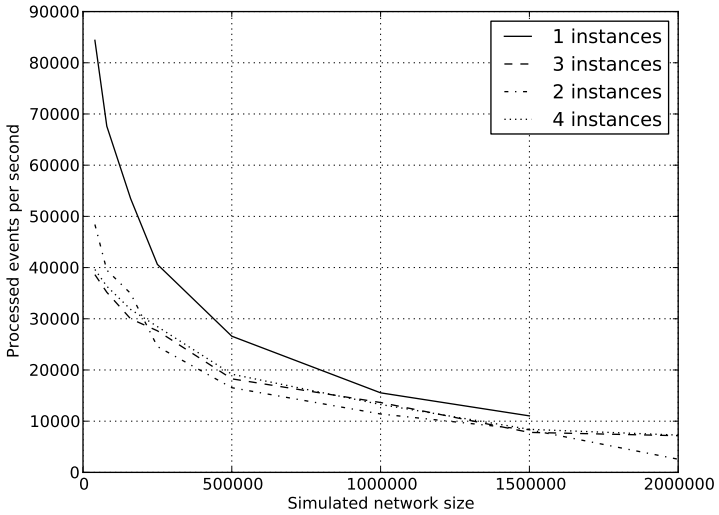


Figure 5.8: Simulation processing speed in each instance with degree=1

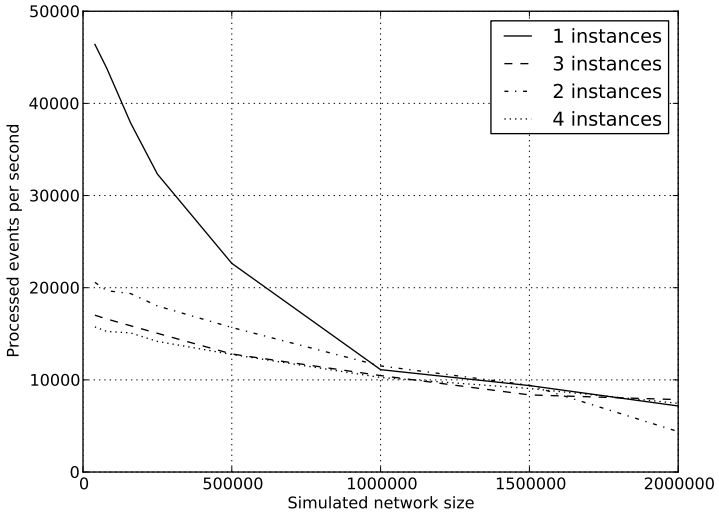


Figure 5.9: Simulation processing speed in each instance with degree=3

The results of this test are displayed in Figures 5.6, 5.7, 5.8, 5.9.

From the four presented graphs, Figures 5.6, 5.7 present the combined event processing speed for all the network. The first graph shows the behavior of the network with an Infection degree of 1, and the second graph shows the same network but with an infection degree of 3.

We can contrast these two graphs and it becomes apparent the dichotomy of the starvation (messages are consumed faster than they are produced) character of the first set of simulations with the over-production present in the second set of simulations.

If we take both graphs and look at the behavior of simulations composed of only one instance, we can see they are quite similar, absolute values differ as an event with degree 3 takes longer to process than an event with degree 1 however, the slope of the curve is very similar, the processing speed is inversely proportional to the size of the simulated network. The simulations only running on one instance, are bottlenecked by the control object iterating over local nodes.

In the starvation scenario we can see that some speedup can be achieved using more instances but the amount of speedup is quite low. In this scenario the number of messages produced are not enough to keep buffers full so that the simulator may run at maximum speed when the control is not running. What we see in this scenario is the same curve caused by the iteration process but slightly faster, taking advantage of the parallelized iteration. We cannot see, however, any significant evidence of parallelized simulation.

This process of parallelized simulation is exactly what we can see, on simulations running in an over-production environment. The number of messages produced is high enough to fill incoming buffers and that makes it possible for simulation to be parallelized along with iteration. The starvation simulations are still overall faster in absolute numbers, because new events take time to create and process, and take up memory that must be allocated and deallocated.

The best information to take from this test is that DIPS is capable of overcoming the limits created by controls that iterate the network, and is doing it by parallelizing both the iterations and the simulation.

5.5 DIPS Distributed Simulation Memory Overhead

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.

5.5.1 DIPS Simulation Memory Usage

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 is Figures 5.10, 5.11. The the first figure we see the memory used by the JVM during each instant of the initialization phase, in the second we see a moving average of the same information, to help the visualization process.

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

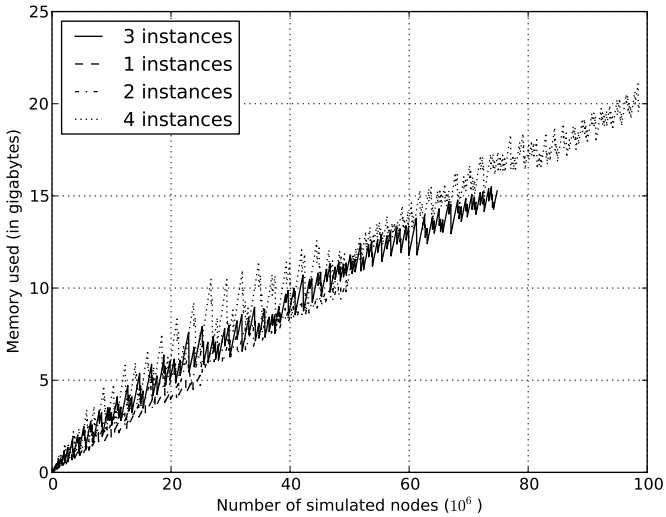


Figure 5.10: Memory used by DIPS as a function of the simulated network size.

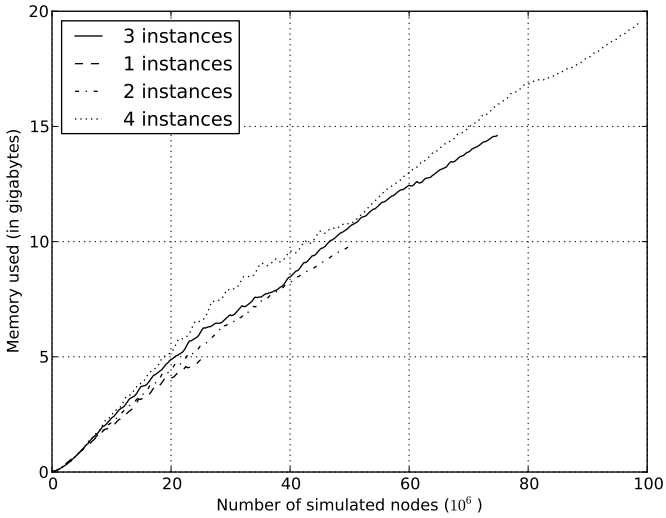


Figure 5.11: Memory used by DIPS as a function of the simulated network size, using a 2000 point moving average.

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 Conclusions

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.

Bibliography

[1] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. 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*, pages 1–14, 2002.

[2] D.P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@ home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[3] A. Andrzejak and Z. Xu. Scalable, efficient range queries for grid information services. 2002.

[4] A.R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. In *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 353–366. ACM, 2004.

[5] N. De Bruijn. A combinatorial problem. In *In Proc. Koninklijke Nederlandse Akademie van Wetenschappen*, volume 49, pages 758–764, 1946.

[6] R. E. Bryant. Simulation of packet communication architecture computer systems. Technical report, Cambridge, MA, USA, 1977.

[7] R. Buyya and M. Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and Computation: Practice and Experience*, 14(13-15):1175–1220, 2002.

[8] M. Cai, M. Frank, J. Chen, and P. Szekely. Maan: A multi-attribute addressable network for grid information services. *Journal of Grid Computing*, 2(1):3–14, 2004.

[9] H. Casanova. Simgrid: A toolkit for the simulation of application scheduling. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 430–437. IEEE, 2002.

[10] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC*, 20:2002, 2002.

[11] K.M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, 5:440–452, 1979.

[12] K.M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(4):198–206, 1981.

[13] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[14] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies*, pages 46–66. Springer, 2001.

[15] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a common api for structured peer-to-peer overlays. 2003.

[16] Y.K. Dalal and R.M. Metcalfe. Reverse path forwarding of broadcast packets. *Communications of the ACM*, 21(12):1040–1048, 1978.

[17] P. Garcia, C. Pairot, R. Mondéjar, J. Pujol, H. Tejedor, and R. Rallo. Planetsim: A new overlay network simulation framework. *Software Engineering and Middleware*, pages 123–136, 2005.

[18] T. Gil, F. Kaashoek, J. Li, R. Morris, and J. Stribling. p2psim, a simulator for peer-to-peer protocols, 2003.

[19] A. Gupta, O.D. Sahin, D. Agrawal, and A.E. Abbadi. Meghdoot: Content-based publish/subscribe over P2P networks. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 254–273. Springer-Verlag New York, Inc., 2004.

[20] S. Hanks, M.E. Pollack, and P.R. Cohen. Benchmarks, test beds, controlled experimentation, and the design of agent architectures. *AI magazine*, 14(4):17, 1993.

[21] RT Hepplewhite and JW Baxter. Broad agents for intelligent battlefield simulation. In *Proceedings of the 6th Conference on Computer Generated Forces and Behavioural Representation*, 1996.

[22] A. Iamnitchi, I. Foster, and D.C. Nurmi. A peer-to-peer approach to resource location in grid environments. *INTERNATIONAL SERIES IN OPERATIONS RESEARCH AND MANAGEMENT SCIENCE*, pages 413–430, 2003.

[23] S. Iyer, A.I.T. Rowstron, and P. Druschel. Squirrel: a decentralized P2P web cache. In *Proc. Annual ACM Symposium on Principles of Distributed Computing, Monterey, CA*, 2002.

[24] N.R. Jennings and M.J. Wooldridge. Applications of intelligent agents. *Agent technology: Foundations, applications and markets*, pages 3–28, 1998.

[25] M. Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal distributed hash table, 2003.

[26] Jon Kleinberg. The small-world phenomenon: An algorithmic perspective. In *in Proceedings of the 32nd ACM Symposium on Theory of Computing*, pages 163–170, 2000.

[27] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. pages 190–201, 2000.

[28] N. Leibowitz, M. Ripeanu, and A. Wierzbicki. Deconstructing the kazaa network. In *Internet Applications. WIAPP 2003. Proceedings. The Third IEEE Workshop on*, pages 112–120. IEEE, 2003.

[29] Y.B. Lin and E.D. Lazowska. *Reducing the state saving overhead for Time Warp parallel simulation.* Dep. of Computer Science and Engineering, Univ. of Washington, 1990.

[30] E.K. Lua, J. Crowcroft, M. Pias, and R. Sharma. A Survey and Comparisons of Peer-to-Peer Overlay Network Schemes.

[31] V. Madisetti, J. Walrand, and D. Messerschmitt. Synchronization in Message-Passing Computers—Models, Algorithms, and Analysis. *Proceedings of the SCS Multiconference on Distributed Simulation*, 22(1):25–48, 1990.

[32] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. pages 183–192, 2002.

[33] Gurmeet Singh Manku, Mayank Bawa, Prabhakar Raghavan, and Verity Inc. Symphony: Distributed hashing in a small world. In *In Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, pages 127–140, 2003.

[34] C. Mastroianni, D. Talia, and O. Verta. A super-peer model for building resource discovery services in grids: Design and simulation analysis. *Advances in Grid Computing-EGC 2005*, pages 132–143, 2005.

[35] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric, 2002.

[36] A. Medina, A. Lakhina, I. Matta, and J. Byers. BRITE: An approach to universal topology generation. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2001. Proceedings. Ninth International Symposium on*, pages 346–353. IEEE, 2002.

[37] A. Montresor and M. Jelasity. Peersim: A scalable p2p simulator. In *Peer-to-Peer Computing, 2009. P2P'09. IEEE Ninth International Conference on*, pages 99–100. IEEE, 2009.

[38] K.L. Morse et al. *Interest management in large-scale distributed simulations.* Citeseer, 1996.

[39] A. Muthitacharoen, R. Morris, T.M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. *ACM SIGOPS Operating Systems Review*, 36(SI):31–44, 2002.

[40] S. Naicken, A. Basu, B. Livingston, and S. Rodhetbhai. A survey of peer-to-peer network simulators. In *Proceedings of The Seventh Annual Postgraduate Symposium, Liverpool, UK*. Citeseer, 2006.

[41] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. *Scalable wide-area resource discovery*. Citeseer, 2004.

[42] A.U. Petra, P. Tyschler, and D. Tyschler. Modeling Mobile Agents. In *In Proc. of the International Conference on Web-based Modeling and Simulation, part of the 1998 SCS Western Multiconference on Computer Simulation. San Diego*. Citeseer, 1998.

[43] C. Greg Plaxton, Rajmohan Rajaraman, Andrea W. Richa, and Andr'ea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. pages 311–320, 1997.

[44] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31:161–172, August 2001.

[45] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a dht. In *In Proceedings of the USENIX Annual Technical Conference*, 2004.

[46] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *Arxiv preprint cs/0209028*, 2002.

[47] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems, 2001.

[48] Antony Rowstron and Peter Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. pages 188–201, 2001.

[49] N. Santos, L. Veiga, and P. Ferreira. Vector-field consistency for ad-hoc gaming. *Middleware 2007*, pages 80–100, 2007.

[50] K. Shudo, Y. Tanaka, and S. Sekiguchi. Overlay weaver: An overlay construction toolkit. *Computer Communications*, 31(2):402–412, 2008.

[51] D. Spence and T. Harris. Xenosearch: Distributed resource discovery in the xenoserver open platform. In *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on*, pages 216–225. IEEE, 2003.

[52] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31:149–160, August 2001.

[53] V.S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: practice and experience*, 2(4):315–339, 1990.

[54] D. Talia and P. Trunfio. Toward a synergy between p2p and grids. *IEEE Internet Computing*, page 96, 2003.

[55] D. Talia and P. Trunfio. Peer-to-peer protocols and grid services for resource discovery on grids. *Advances in Parallel Computing*, 14:83–103, 2005.

[56] J. Vogel and M. Mauve. Consistency control for distributed interactive media. In *Proceedings of the ninth ACM international conference on Multimedia*, pages 221–230. ACM, 2001.

[57] H. Yu and A. Vahdat. Building replicated Internet services using TACT: a toolkit for tunable availability and consistency tradeoffs. In *Advanced Issues of E-Commerce and Web-Based Information Systems, 2000. WECWIS 2000. Second International Workshop on*, pages 75–84. IEEE, 2002.

[58] Ben Y. Zhao, John Kubiatowicz, Anthony D. Joseph, Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical report, 2001.

[59] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. pages 11–20, 2001.