Distributed Peer-to-Peer Simulation

Plataforma de partilha de CPU para simulação de protocolos de comunicação Peer-to-Peer

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

Instituto Superior Técnico

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

1 Introduction

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

Peer-to-peer

Peer-to-peer (P2P) systems are distributed computer systems where network communication is done directly between endpoints, not requiring a central server as an intermediary. They oppose common client-server architecture that composes the large majority of network communicating systems today.

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

Historically P2P systems have existed since the beginning of network communication. The rise of the World Wide Web brought the need for specific endpoints, more powerful than regular endpoints, capable of high throughput and with well-known names. These endpoints were then called servers as the WWW model required them to **serve** pages, and later, requests in general. This was the rebirth of the client server model (created with remote procedure calls), which proved to be a simple, efficient and secure model, servers were assumed to always work, clients could rely on them for every need.

Another shift occurred in the network communication paradigm when hardware became powerful enough to store and reproduce compressed music files. Huge collections previously only present in materialized media (such as in CDs) could then be dematerialized, stored in a computer hard drive and **shared**. Sharing proved to be difficult, using client-server paradigm, sharing music files would clog network links to the server and/or fill the servers storage space very quickly. The solution came in the form of

a service called Napster [1]. Users would download a program that would register itself, along with the files to be shared, in a server. The program would also browse that server for the files other users had registered. The ground breaking feature was that actual transfer of the files would occur between the endpoints, or peers, and communication with the server was required only for registering, browsing and querying.

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 huge 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.

Peer-to-peer simulation

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

Current limitations and possible solutions

Current peer-to-peer simulation suffers from a peer-count limitation due to memory usage. 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, to remove the memory limit and still execute the simulation with acceptable performance.

Simulation environments such as Peersim [37] have a limit of nodes they can simulate. This limit could be overcome with a distributed Peersim. Maintaining API compatibility at the cost of a performance penalty, a distributed Peersim would allow the developer to simulate more nodes in his protocol with no extra development.

A distributed Peersim would allowed developers to reach a much larger network size.

Parallelized simulation

In a very large simulated network, even a small performance penalty could have a damaging effect on the development cycle. Peer-to-peer simulation is by definition full of parallelizable tasks. By taking advantage of the inherent parallelism the developer could use Peersim communication APIs to parallelize the simulation. Parallelism could be achieved at the cost of no central control and small changes to the code.

A simulation using a distributed Peersim with distributed control could result in a significant speedup.

[1] http://www.napster.com

Event-driven simulation

Cycle driven simulation is mostly suited for early development as it may hide problems with the protocol, due to its serialized nature. A developer looking for a more realistic simulation using event-driven simulation would be able to take full advantage of a distributed Peersim. The encapsulation necessary to handle events in the simulation is in itself parallel, therefore suitable to a distributed simulation environment. The amount of parallelism could be configured from globally synchronized to fully parallelized message delivery, allowing a very flexible simulation code.

Simulation using a distributed Peersim event engine would allow great flexibility without limiting the amount of simulated nodes.

2 Objectives

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

We will address some of these problems by creating a distributed peer-to-peer simulator.

In particular, this thesis aims to fullfil the following objectives:

1. Overcome memory limitations of current peer-to-peer simulators by distributing the simulation, in a parallelized fashion, over a number of nodes.
2. Reach one order of magnitude higher peer-count than common max peer-count (one million).
3. Maintain maximum compatibility with the chosen implementation platform.
4. Minimize overhead on simulation processing time per peer.
5. Implement, maintain and document distributed realistic simulation (event driven) with similar metrics to simple (cycle driven) simulation, regarding performance and memory footprint.

3 State of the Art

3.1 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.

3.1.1 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.

3.1.1.1 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.

3.1.1.2 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 3.1.2). 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.

3.1.2 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.

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.

	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 1. Comparison of structured peer-to-peer protocols.

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.

3.1.3 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, 55, 34].

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.

3.2 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).

3.3 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.

3.3.1 Ns-2

Ns-2 [2] 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.

3.3.2 Peer-to-peer Network Simulation

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

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 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.

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

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.

3.4 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.

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. Comparison of Peer-to-Peer Simulators

Event An instantaneous occurrence that might change the state of the system.

Processes are the autonomous components to be ran 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.

3.4.1 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.

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].

3.5 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 it processes only access information relevant to them.

4 Architecture

This thesis is part of a larger internal project (see Figure 1) to:

- Increase scalability of simulation tools
- Unleash the power of idle CPU cycles
- Further Grid and overlay research

The project aims at solving not only the limitations of current peer-to-peer simulation, but to create an environment for peer-to-peer overlay and Grid research that is extensible, promotes code reuse and defines new standards. Leveraging idle CPU cycles that are so prevalent today, as more and more computer-like devices are present in peoples life, is both the goal and the reasoning behind this project as applications must be built to take advantage of this reality. This thesis is a step towards improving the peer-to-peer simulation tools and building the distributed basis that will allow future grid research to take advantage of the global network environment.

4.1 DIPS: Distributed Implementation of the Peersim Simulator

Peersim is a very flexible and simple tool that allows for the same idea to be implemented in a number of different ways. In particular, simulations can be implemented using both a cycle driven engine and an event driven engine.

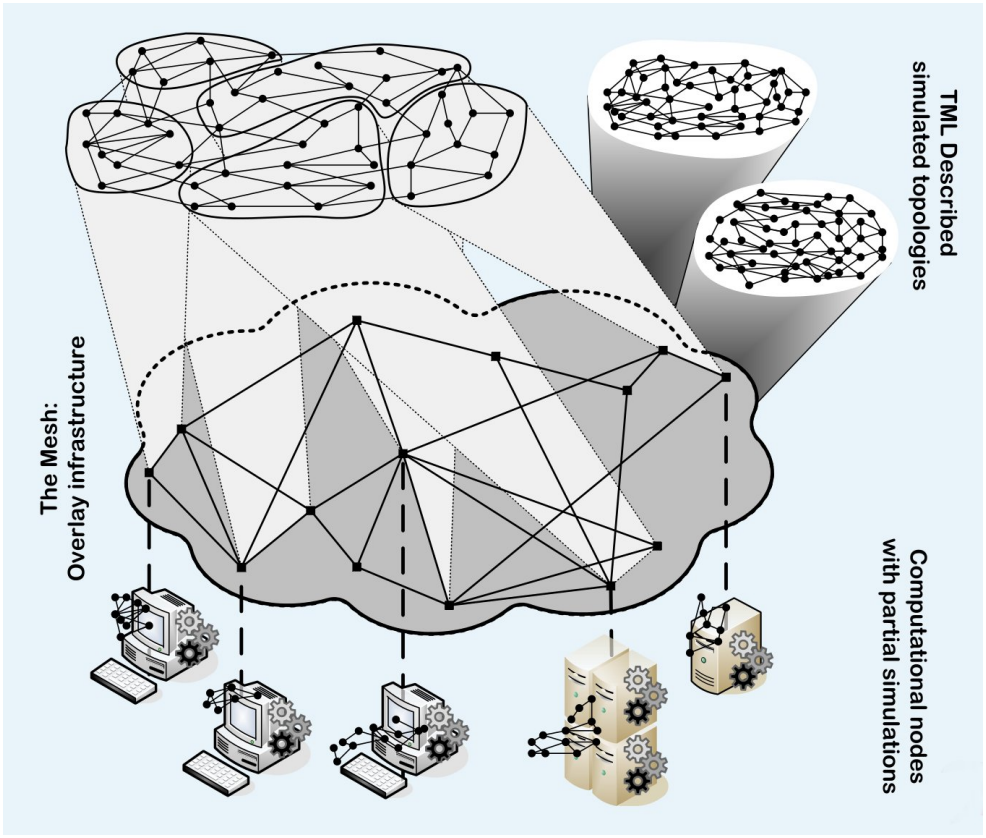

Fig. 1. Overview of the Peer4Peer architecture

Peersim seems to focus on simplicity and takes a *get out of the way* approach. One example of this is the direct access to objects simulating the peers, these objects have access to the entire simulation state and could potentially create a web of references not at all appropriate for distributed execution.

Taking advantage of the flexibility in Peersim, we propose DIPS, a distributed implementation of the Peersim simulator. As shown in Figure 2, DIPS is a system composed of several Peersim instances communicating between them to improve the amount of resources available to the simulation.

We propose the execution engine of DIPS to be divided in three complementary aspects, to be implemented in sequence. Peersim current implementation is already divided between a cycle-driven mode and a event-driven mode. We maintain these two modes and aim at maintaining compatibility with the current API; we add a third mode in order to, although breaking compatibility, overcome the performance issues that the cycle-driven mode will unavoidably suffer.

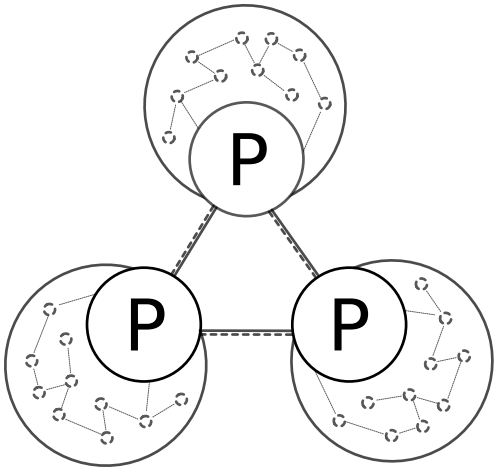

Fig. 2. Overview of DIPS instances

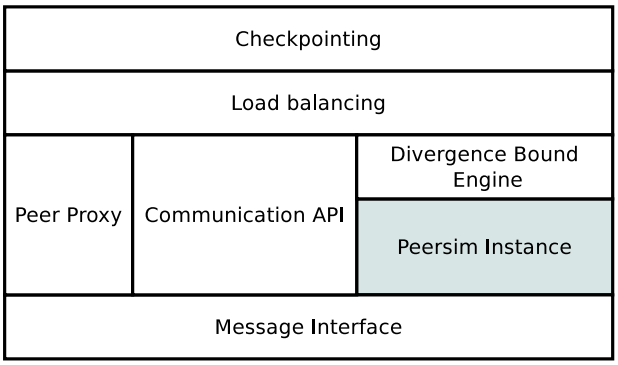

Fig. 3. Overview of the DIPS architecture

4.2 Aspect I: Fully transparent distributed interface

This approach is the one that most resembles original Peersim implementation. Its main purpose is to maintain compatibility with simulations built for the original Peersim. As a result it aims to provide one simple improvement: developers wanting to implement a simple simulation, or unwilling to deal with the specific issues of a distributed simulation, can use this approach to overcome Peersim original memory limitation using all of the original API.

However some limitations are inevitable. Peer proxies are used to communicate between central control and distributed peers, therefore only methods can be called, central control does not have access to properties of the peers. Also any parameter and return value of a *proxied* method, must be serializable. The developer must write the implementation so that modification to objects directly under a peer control, be encapsulated.

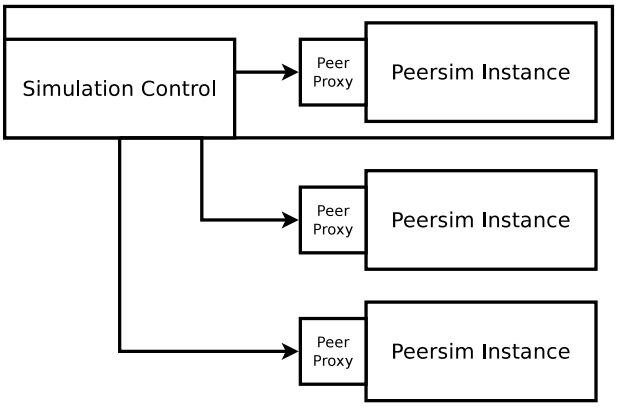

Fig. 4. Aspect I: Fully transparent distributed interface

4.3 Aspect II: Distributed Control with aggregation

This approach would be the preferred way to take advantage of DIPS in the cycle driven mode. The objective is the same as the first approach, overcome the memory limitation of a single Peersim instance. In order to do so and keep performance at acceptable levels, a set of APIs would be offered. The original API would be respected, however it would only have access to local peers. Communication between control objects in different Peersim instances would be guaranteed through an additional API, including transfer of peers between instances.

The figure shows that simulation control would be distributed among instances. This control would have to synchronize or, at least operate knowing that each piece only controls local peers. This aspect requires that developers know their simulation will be ran in a distributed environment.

4.4 Aspect III: Event driven distributed simulation

The event driven simulation aspect would be the primary approach for event driven simulation. Messages to be passed between nodes are intercepted and routed to the right instance. The api would use a best effort approach to the message transport delay simulation. All compatibility between the event driven engine and the cycle driven engine would be broken. This is necessary to limit the overhead of synchronization between Peersim instances. Control objects can be used as in the second aspect.

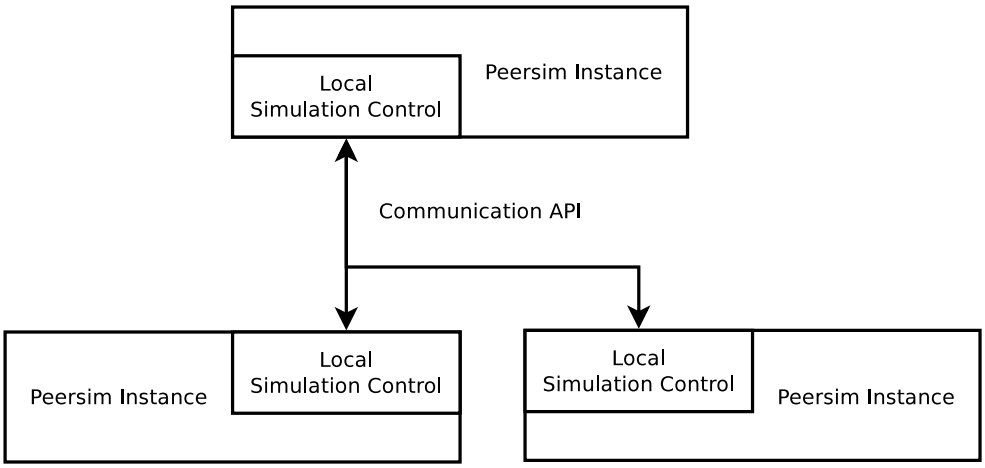

Fig. 5. Aspect II: Distributed Control with aggregation

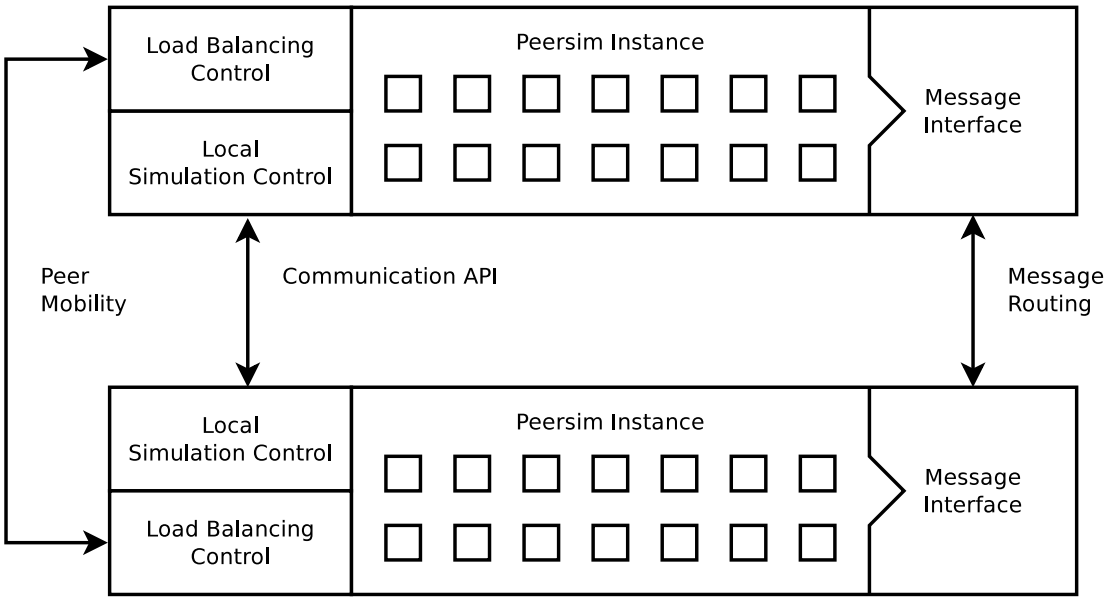

Fig. 6. Aspect III: Event driven distributed simulation

4.5 Simulation Realism

Determinism is not a requirement for peer-to-peer protocols in live deployments. As such, it is also not a strict requirement for peer-to-peer simulation. Reproducibility would be a welcomed characteristic to the simulation engine, unfortunately much of the gains of parallelization are incompatible with the determinism required for reproducibility.

Still, we want the simulation to portray the simulated system/overlay as realistically as possible, according to the expected protocol behavior when deployed in a wide area network.

Peer-to-peer protocols, contrary to lower-level network protocols, cannot rely for correctness on guarantees regarding message latency or delivery delay. Therefore, **correct protocols will operate correctly on a distributed simulator without such guarantees**.

Nevertheless, we want simulation to be realistic, that is, that the observed node behavior and interactions among nodes are coherent with what would be observed in a real deployment assuming normal network operation.

The simulation of each partition without any coordination would reveal isles in the overlay, not reflecting any characteristic of the topology, but simple artifacts of the simulation. Unlike in real deployment, regions would be created where message links with a much higher throughput and much lower latency would be formed between nodes present in these regions. The presence of these links would generate an artificial increase of interaction between the nodes connected by them. This would cause imbalance in resource usage and possibly lower resource utilization in overlays characterized by great dynamism (e.g., resource discovery in a cycle-sharing system).

We intend to offer a spectrum of realism levels in the simulation, trading better performance for increasing unrealistic deviations in relative message delivery delay:

- On one end, all instances implement a barrier where they can only progress when they achieve confirmation that there are no outstanding messages to be delivered to nodes simulated in that instance, across the simulation.
- On the other end, simulation will carry on as described above, for those protocols that do not suffer from message delivery time varying unrealistically.

Between these two extremes, inspired by approaches based in divergence bounding [49, 57], instances may delay and batch messages sent to nodes in other instances (i.e. transfer messages in batch packets) up to a maximum divergence bound, upon which, confirmation of message delivery is mandatory (i.e., a barrier is dynamically set-up among two or more instances of PeerSim).

The divergence bound may be expressed as a real time simulation parameter (e.g., every 10 seconds of clock time) or as simulation time (e.g., up to 100 messages between nodes placed in different instances).

4.6 Routing and Load Balancing

Simulated peers must be distributed through the simulator instances. This poses two problems:

- The number of peers must be balanced throughout the instances.
- The simulation engine must know in which instance, any given peer, is located (i.e., being simulated).

We propose two solutions, a distributed hash table and neighborhood clustering.

The traditional approach is to use a distributed hash table. Simulated peer *ids* are already generated randomly, we apply a consistent hashing algorithm to these *ids*, this will give the location of the peer with that id. By using consistent hashing we guarantee load balancing throughout the instances. Location of peers is therefore only dependent on the hashing algorithm, with no memory overhead.

In order to optimize peer location we propose that this location can be defined by clustering together neighboring peers. Peer permanent location will only be decided when the implementing protocol defines a neighborhood between that peer and other peers. Then the peer temporary *id* will be substituted by an *id* prefixed with a *namespace* that identifies the peer location. It is noteworthy that the peer *id* will maintain its random properties as the second part of the *id* will still be random.

4.7 Fault-Tolerance

When PeerSim fails, simulations must be restarted. With several instances of PeerSim, the probability of one failing increases. Whenever instances block on a global barrier (e.g., every 2 hours of execution), each one saves on disk:

- its node list and state
- message queue
- message batch packets unsent and those received but not yet inserted in queue
- global simulation time

If one fails, the whole simulation can be restarted from a previously completed checkpoint.

5 Assessment

In order to execute a general assessment of our peer-to-peer simulator we will adapt, as necessary, the protocols distributed as examples with Peersim, to work with our distributed peer-to-peer simulator. All necessary adaptations will be analyzed to guarantee their necessity. Our goal is to provide an API that gracefully degrades, i.e. adaptations are not mandatory but they will improve performance.

A number of tests will then be executed to assess the correctness and performance of the simulator:

- Each protocol must be tested in both the original Peersim and in our simulator to guarantee the simulators correctness, i.e., that the protocol does not break by having simulated peers with wrong or unrealistic behavior. Furthermore, any protocol that exhibits some kind of convergence property (e.g., estimating network radius) will also behave accordingly in simulation.
  This does not entail necessarily complete determinism (in message delivery order) among different runs of the same simulation. This stricter propoerty is not required by peer-to-peer protocols as it is not enforceable in real world wide area networks.
- Each protocol must also be tested in increasing peer count, in both Peersim and DIPS to gain empirical data about the performance penalty incurred by our distributed simulator.

The distributed peer-to-peer simulator must be evaluated in accordance with objectives described in section 2. Tests will be performed to assess the completion of those objectives:

Overcome memory limitations of current peer-to-peer simulators We will conduct independent experiments to assess the memory limitations of current peer-to-peer simulators using provided example protocols. These protocols will then be adapted to work with our distributed peer-to-peer simulator in order to prove those memory limitations have been overcome.

Reach one order of magnitude higher peer count The maximum peer count is claimed by the Peersim simulator, which will serve as the basis to DIPS, and is set at one million nodes. We will create a simple protocol for our simulator to prove it can reach ten million simulated nodes.

Minimize simulation overhead We will further assess the performance gains of the second aspect of our simulator by comparing absolute simulation times of similar simulations in both the first and second aspects, therefore evaluating simulation speedup in throughput and, ultimately, simulation scalability.

6 Conclusions

Peer-to-peer overlays and applications are very important in current day-to-day applications. In the future this seems to be even more relevant.

In this thesis we have shown the benefits of using peer-to-peer technologies. We have also shown the importance of simulation in the development of peer-to-peer applications and protocols.

Unfortunately current peer-to-peer simulators are flawed and unable to serve their purpose beyond the limitations in memory and performance of a single machine.

We concluded that Peersim is the most complete simulation tool in the peer-to-peer realm, it's cycle-based and event-driven engine allow developers access to the different levels of abstraction needed in each step of the development process.

We proposed DIPS, a distributed implementation of the Peersim simulator to overcome these limitations. By utilizing technologies from parallel systems simulation, distributed agent simulation and peer-to-peer overlays themselves we hope to produce a tool that not only overcomes the limitations of current peer-to-peer simulators but also outperforms them.

References

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.

A Planning

The implementation of the DIPS simulator will be executed according to the following plan:

- Prepare the Peersim engines for distributed execution - 4 weeks
- Implement the proxy component - 2 weeks
- Implement communication APIs for distributed control - 3 weeks
- Implement the message routing interface - 1 weeks
- Implement the checkpoint component - 3 weeks
- Implement the message buffer component - 1 week
- Implement the optimistic algorithms to the message buffer component - 4 weeks
- Implement the neighborhood based load balance component - 3 weeks
- Final tests - 3 weeks

This project has an expected duration of 24 weeks, to be started on 01-02-2011 and end at 19-07-2011.