Title:	Evaluation of Replication techniques for P2P Systems
Volume:	1/1
Student:	Alina-Diana Potlog
Director/Ponent:	Fatos Xhafa Xhafa
Department:	Llenguatges i Sistemes Informàtics
Date:	21st June, 2011

EVALUATION OF REPLICATION TECHNIQUES FOR P2P SYSTEMS

Alina-Diana POTLOG[1], Fatos Xhafa[2]

Contents

[1]Faculty of Automatic Control and Computers, University "Politehnica" of Bucharest, România, e-mail: dianapotlog@gmail.com

[2] Universitat Politècnica de Catalunya , Barcelona, Spain, e-mail: fatos.xhafa@gmail.com

1. Introduction

Replication techniques are commonplace in P2P computing systems. Initially, the objective of such techniques has been to ensure *availability* of information (typically files) in P2P systems under highly dynamic nature of large P2P systems. For instance, in P2P systems for music file sharing, the replication allows to find the desired file at several peers as well as to enable a faster download.

It should be noted however that in the above, the files or documents are considered static, that is, they don't change over time (or, if the change, new versions are uploaded at different peers). In a broader sense, however, the documents could change over time and thus the issues of availability, consistency and scalability arise. Consider for instance, a group of peers that collaborate together in a project. They share documents among them, and, in order to increase availability, they decide to replicate the documents. Because documents can be changed by peers, for instance different peers can edit the same document, and thus changes should be propagated and made to the replicas to ensure consistency. Moreover, the consistency should be addressed under the dynamics of the P2P systems, which implies that some updates could take place later (as the peer might be off at the time when document changes occurred).

Replication can be seen as a family of techniques. Fundamental to any of them is the degree of replication (full vs. partial), as well as the source of the updates and the way updates are propagated in the system. Its is therefore interesting to study different replication techniques to see their performance in the context of a peer-group.

All P2P replication systems rely on a P2P network to operate. This networks is an overlay network, meaning that it is built on top of the physical network. Section 3 presents P2P networks in depth. Section 4 presents the JXTA protocol which is a popular P2P protocol which allows any device connected to a network to exchange messages and collaborate independently of the underlying network topology.

We analyze the existing replication techniques in Section 5 and propose a replication system for asynchronous collaboration, with real-time delivery of updates, having a super-peer network and using optimistic replication techniques (Section 6). Our proposed replication system is suited for both full and partial replication. Section 7 details the implementation of our system and the experimental study can be found in Section 8. We conclude the results in Section 9.

2. Objectives and Context

We plan to build a replication system for asynchronous collaboration, with real time delivery of updates, having a super-peer network and using optimistic replication techniques. The scope of replication is among the peers that are part of the collaborative group. The replication system must allow late joining of peers. This means that, when a peer connects to the group, its replicated documents arrive at a consistent state with the other replicas across the system.

The context of the project is that of P2P systems, in which a group of peers work together to accomplish a common project (hereafter "group-project"). Peers share documents, tasks and project information among them. We will make the following assumptions:

- P2P network has a super-peer structure. Peers join *separated* groups, each group having its own super-peer (Illustration 1)
- Super-peers and peers are assumed to have *Rule Engines*. A rule engine will be used here to implement the replication schemes. A peer rule engine might state that only consecutive versions of received updates must be immediately enforced at local site while the other versions might be queued for late enforcing.
- Super-peers and peers are also assumed to have storage (standard configuration –no special requirements) and software (e.g. Java Virtual Machine, DB2 or Oracle).
- A peer-group is assumed to work on the completion of a group-project, in which, each peer has to accomplish some tasks (tasks of the projects are distributed to different peers in advance –as input data– see later).
- Each peer stores locally its *proper* documents, that is, documents related to its tasks as well as replicas of others peers' documents.
- The workflow of the project is associated with a fine state machine, from which can be known, at any time, the status of tasks of the project, the task precedence, and from which can be monitored the project progress as well.

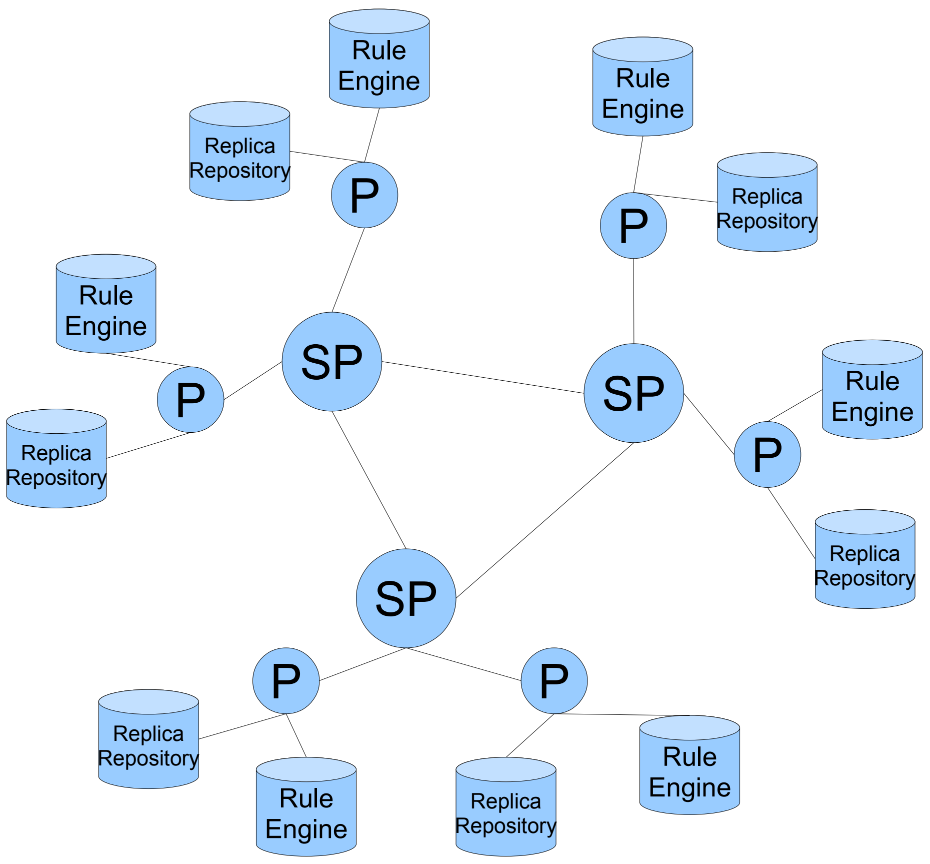


Illustration 1: P2P super-model

3. P2P Networks

P2P systems rely on an abstract overlay network on top of the native or physical network topology. This abstract network is built at Application Layer. Content is typically exchanged over the underlying Internet Protocol (IP) network. The degree of centralization and the topology of the overlay network influence the non-functional properties of the P2P systems (e.g. fault-tolerance, performance, scalability, security). There are two main classes: unstructured and structured.

In the structured P2P networks, peers and resources are organized following specific criteria and algorithms, which lead to overlays with specific topologies and properties. These type of networks usually implement distributed hash table-based (DHT) indexing.

Unstructured P2P networks do not use any algorithm for organizing or optimizing network connections. There are three models of unstructured P2P networks: *centralized, decentralized* and *super-peer*.

The following subsections describe the above taxonomy of P2P networks and stress the benefits of using super-peer networks in document replication.

3.1. Structured P2P Networks

Structured P2P Systems control the overlay topology and data placement and thus ensuring that any node can route a search to a peer that has the specified file, even if this is a rare one.

Distributed Hash Table (DHT) is the main representative of structured P2P systems. It provides a look-up service very similar to a hash table. DHT holds pairs (key, value) and any node in the system can find the value associated with a certain key. A key is an object identifier, while a value represents the object contents. Moreover, a peer keeps a list of other peers (neighbors) and a routing table that associates its neighbors' identifiers to the corresponding addresses. Mappings from keys to values are distributed across the nodes thus a change in the dynamic of the system causes a minimal amount of failure (Illustration 2). For this reason, DHT achieves scaling to extremely large number of nodes and handles situations when nodes join, leave or fail.

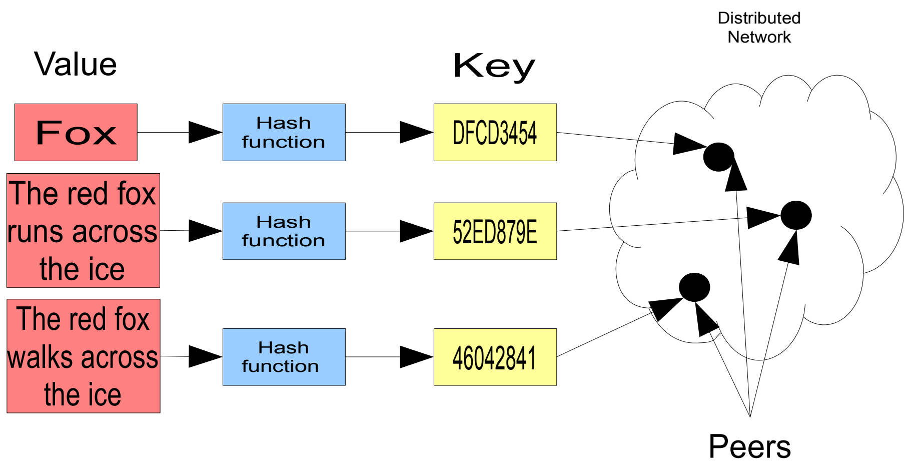


Illustration 2: Distributed hash tables

A query can be routed efficiently in *O(logN)* hops, where N is the number of nodes in the system. Peers have limited autonomy, as they have to store the values corresponding to a certain range of keys. Furthermore, DHT queries are limited to exact match keyword search although but there are new solutions that extend DHT capabilities to handle general queries like range queries and join queries [9].

Notable structured P2P systems include Bit Torrent, Chord, PIER, P-Grid, CAN [8].

3.2. Unstructured P2P Networks

Unstructured P2P systems have the overlay network established in an arbitrary manner and the data placement doesn't take into account the overlay topology. When a peer wants to find a piece of data in the network, the search query has to be flooded through the network until the desired data is located. Though this mechanism is simple, it is expensive as it causes high amount of signaling traffic in the network. Moreover, search success is essentially about coverage. Popular content can be found at many locations and any peer searching for it is likely to find the same thing. On the contrary, if a peer is looking for rare data shared by only a few other peers, then there is a great probability that search will be unsuccessful. Most of P2P networks are unstructured.

Unstructured networks may be centralized or decentralized. The following subsections present them in some more detail.

3.2.1. Centralized Networks

The reference application for centralized networks is *Napster*. *Napster* hosts act as both client and server for the exchange of musical content. A host first joins the network by connecting to a central website (broker) that holds file directories for all the hosts in the network. Once connected, the host then transmits to the broker meta-data information containing all the music files it serves. When a host wants to retrieve a music file, it queries the broker which in turn answers with a list of peers that have the required file. Ideally, this list is sorted based on the closest/least-loaded peers. The client can then coordinate with the broker on the exchange of the desired file from one of the remote hosts. In Illustration 3, peer D is downloading a file directly from peer B. The broker only participated in searching for the file and setting up the file exchange, but the peer D downloads the file directly from peer B.

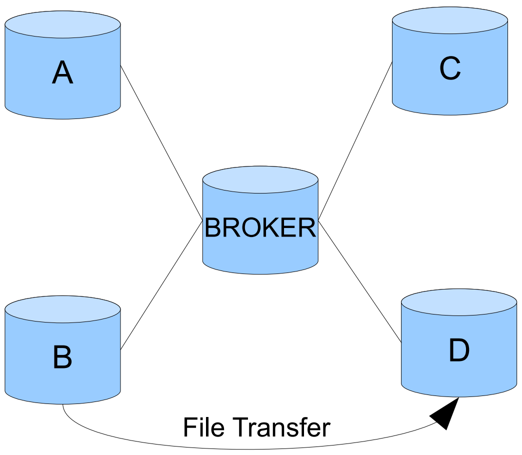


Illustration 3: Napster Network: the broker coordinates the file transfer among two peers

The advantages *Napster* presents are that it is simple and easy to implement sophisticated search engines on top of the index system. The disadvantages are that it doesn't scale too much and that it has a single point of failure.

3.2.2. Decentralized Networks

In decentralized networks the entire network consists only of equal nodes. As there are no nodes with special infrastructure network, there is just one routing layer. *Gnutella* and *Freenet* are examples of decentralized networks.

Gnutella has no central server for coordination. A *Gnutella* host joins a network by connecting to at least one other node in the network. A client finds certain files by sending requests to all neighbors which in turn will recursively forward the request. The request message has a time to live (TTL) field to ensure that a message does not last forever. When a peer has the files that match the request it sends a response message back along the path the request came. Once the client receives the response then it can directly connect to the peer that has the desired files and download them. For example, in Illustration 4, client A directly downloads the file from peer F.

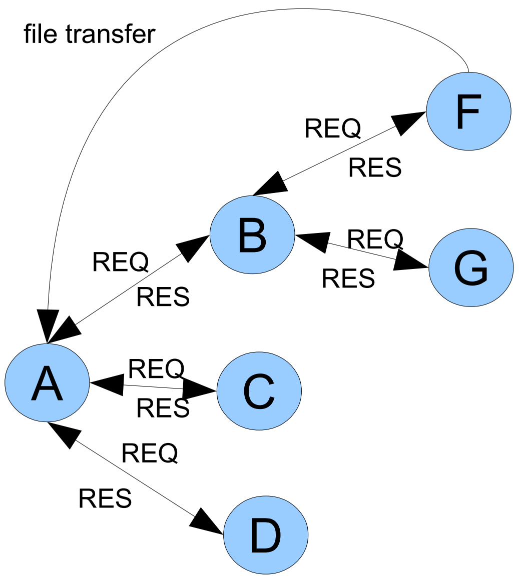


Illustration 4: Gnutella network

Gnutella has the advantages of being robust as it is totally decentralized. But has the disadvantage of not being scalable (the entire network can be swamped with requests) and of being hard to use on slow clients (if broadcast traffic exceeds 56kbps, modem users are cut off).

Freenet is a decentralized system with routing tables. Files are stored according to associative keys. It is like Napster, but better as it tries to cluster information about similar keys. *Freenet* is highly survivable, as the internal processes are under anonymity and decentralized across the network. Information stored on Freenet is distributed around the network and stored on several different nodes. Files and filenames are encrypted. Cannot tell which files are stored on a given node and cannot tell which files are requested by a client. This provides anonymity to the clients and also makes it difficult to censor specific content.

Freenet works as a highly distributed cache, meaning that not only transmits data between nodes, but evenly stores the data. Thus, some amount of disk space is allocated for storing data. *Freenet* also provides distributed storage. A file is split into multiple small blocks with additional blocks to provide redundancy. Each block is treated independently, thus allowing a file to have its parts stored on different nodes. A user shares a file by inserting it into the network. After the process of insertion is finished, the publishing user is free to shut down as the file is already stored in the network. There is no single node responsible for the content, as its replicated to different nodes. This brings two important advantages: high availability and anonymity. On the other hand, no node is responsible for the chunks of data it stores. If a piece of data is not retrieved for a long time and the node holding it is receiving new chunks of data, then the old chunk is removed when the disk space is full.

Freenet network topology is a graph. When a node comes online, it attaches itself to other arbitrary nodes. The network has end-user nodes, route nodes and storing nodes. End-user nodes request and present documents to human users, route nodes only route data and storing nodes actually hold the data. All nodes are identical, there are no dedicated clients or servers.

Files in *Freenet* are encrypted with a public key computed from the name of the file. The providers make public the original name of the files in a public forum (web pages, forums, web spiders search engines). When a consumer asks for a certain file, it first acquires or computes the public key and asks the nearest *Freenet* node for a copy of the file with the computed key. The request has a time to live field (TTL). To locate the file, a depth-first search is used. If the file is found, it is cached at the consumer's node (Illustration 5).

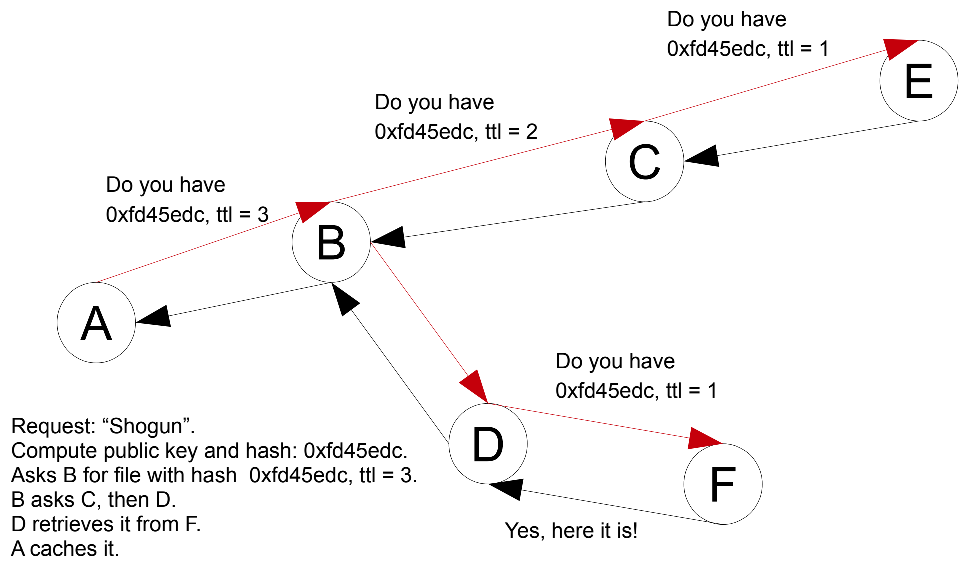


Illustration 5: Freenet query

3.2.3. Super-peer Networks

Super-peer networks are hybrid between centralized and decentralized networks. Super-peer networks contain super-peer nodes that act as a centralized server to a set of clients and as an equal towards the other super-peer nodes [10]. Clients are connected to at least one super-peer. Super-peers have complex tasks like indexing, query processing, access-control, meta-data management (Illustration 6).

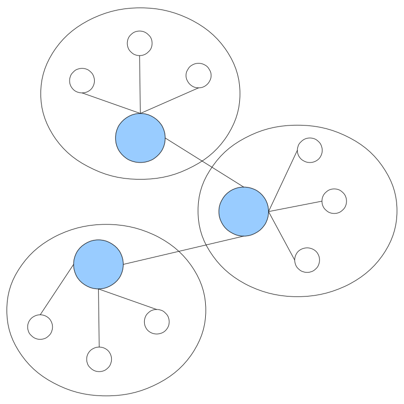


Illustration 7: Super-peer network

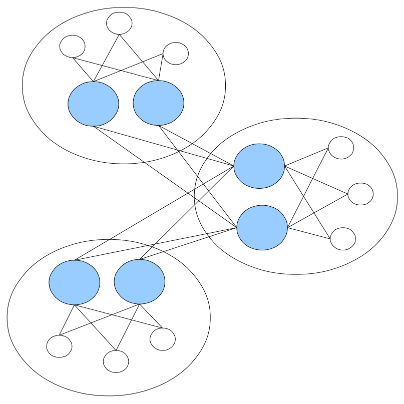


Illustration 6: Super-peer network with 2-redundancy

Super-peer networks have the advantages of a centralized search in terms of efficiency and those of distributed search in terms of autonomy, load balancing and robustness to attacks. Thus, since the super-peer acts as a centralized server for its clients it can handle requests more efficient than each client could. Moreover, as there are several super-peers in the network, a super-peer does not have to face the problem of overloading nor will it become the single point of failure for the entire system. Super-peers can be chosen dynamically based on their computing and storage capacities and replaced in case of failure.

A super peer holds an index with all the data belonging to the peers connected to it. When a client joins the network, it will first send metadata over its collection of files to the super-peer. Peers send requests to the super-peer which in turn processes them by finding the relevant peers directly looking up its index or indirectly using its neighbor super-peers.

A super-peer together with the peers connected to it form a cluster. The size of the cluster is the number of nodes in the cluster, including the super-peer itself. Though a network of clusters is efficient, a super-peer becomes a single-point of failure for its cluster. When a super-peer is down, the nodes attached to it will be off temporarily, till they will find another super-peer to connect to. To minimize the load on the super-peer and prevent failure, redundancy is introduced in the design of the super-peer. It is said that a super-peer is *k-redundant* if there are *k* nodes sharing the super-peer load forming a virtual super-peer. The nodes in the virtual super-peer have equal responsibilities: each node is connected to every client and stores all the clients metadata (Illustration 7).

3.3. P2P Networks and Replica Consistency

When designing a P2P replication system, the chosen P2P network model has a great impact on the performance and load of the system. Thus, it is necessary to analyze each P2P network model taking into account the final scope of replication.

Structured P2P networks use hash tables for data placement which prove to be useless in full replication systems as each site holds replica of the same object.

Picking a model from unstructured P2P systems comes as the right solution.

Centralized P2P networks have the advantage that replication of documents is coordinated only by a single node (master or broker) and thus parallel updates are easy to implement. But, this requires the master to establish a lot of connections with the peers for sending them update notification and furthermore this solution doesn't scale as the master can get overloaded. Achieving consistency in decentralized networks is very hard, as the whole system would be flooded with update notifications. The super-peer model is more appropriate as can achieve consistency more rapidly for small groups and can also scale due to communication between super-peers.

4. JXTA P2P Protocol Specification

This chapter describes JXTA P2P Protocol which we have used to build our P2P network.

JXTA (Juxtapose) is a set of open-source peer-to-peer protocols originally proposed by Sun Microsystems in 2001 that allow any device connected to a network to communicate and collaborate independently of the underlying network topology. JXTA protocols are defined using XML messages and thus programming language independent. Implementations are available for Java SE, C/C++, C# and JavaME. The Java version is though the most advanced implemented.

4.1. JXTA Concepts

Peers

A peer is any type of device (sensor, PDA, supercomputer, virtual process) connected to the internet and that implements one or more of the JXTA protocols. A peer can be composed of multiple physical devices that cooperate or multiple peers can run on a single physical device. Each peer has a unique Peer ID that identifies the peer in the network and each peer runs independently and asynchronously from all other peers in the network.

Peers communicate directly with one another through *endpoints*. A peer endpoint identifies the peer network address which is published by the peer using advertisements. Direct communication between two peers isn't always possible due to physical network boundaries. For example, the two peers can be in different networks, Ethernet and Bluetooth respectively, or behind NAT, firewalls or proxies. Intermediate peers are then used to route the messages between a peer and another.

JXTA peers are classified into three types:

- *Minimal-Edge Peer*. These peers implement only the required core JXTA services and rely on other peers to act as their proxy for other services in order to fully participate in the JXTA network. Examples of minimal-edge peers are sensor devices and home automation devices.

- *Full-Edge Peer*. These peers implement both standard and core JXTA services and can participate in all the JXTA protocols. These peers represent the majority of peers on a JXTA network and include phones, PC's, servers etc.
- *Super-Peer*. These peers have special capabilities, they implement and provide resources to support the deployment and operation of a JXTA network. There are three types of super-peers.
    - *Relay*. Allows peers that are behind firewalls or NAT to take part in the JXTA network. Relay peers use protocols which can traverse the firewall, like HTTP.
    - Rendezvous. Maintains global advertisement indexes and assists edge and proxy peers with advertisement searches. Also handles message broadcasting.
    - Proxy. Provide support to minimal-edge peers so that these can be able to access all the JXTA network functionality.

Peer Groups

A peer group provides the logical clustering of peers. Peers organize themselves into groups. A peer group is uniquely identified by a peer group ID. Each peer group is responsible for creating its own membership policy (from open to highly secured).

Peers can be part of multiple groups simultaneously. Every peer is member of the default group, called *NetPeerGroup*, but it can join other groups at any time.

Network Services

Network services are used by the peers in the JXTA network to collaborate. In order that a network service may be used by the peers, it first needs to be published, then discovered and invoked. There are two types of network services:

- *Peer Services*. A peer service is accessible only on the peer that is publishing that service. If the peer is down, then the service becomes unavailable. Multiple instances of the service can be deployed on different peers, but each instance publishes its own advertisement.

- *Peer Group Services*. A peer group service consists of a collection of service instances, possibly communicating with one another, running on multiple peers of the group. If a peer is down, the collective service still runs provided that the service is available from at least another peer member.

Messages

JXTA Services and applications communicate through JXTA messages. A JXTA message is the basic unit of data exchanged between peers. Each JXTA protocol comprises of a set of messages that are exchanged by the participating peers. Messages are sent between peers using services like *Endpoint Service*, *Pipe Service*, *JXTA Socket*, etc.

There are two types of messages that the JXTA protocols define: binary and XML. These represent the data format used to exchange messages between peers. An election between the two must take into account the characteristics of the underlying network transport.

Pipes

Pipes are an asynchronous, unidirectional and non-reliable message transfer mechanism used for communication and data transfer. They are virtual communication channels used to send any type of data between peers that might not be directly connected through a physical link. A pipe has two endpoints, input pipe and output pipe respectively. JXTA offers three types of pipes:

- *Point-to-point Pipes.* This type of pipe connects two pipe endpoints together, the input pipe of one peer to the output pipe of the other peer. It is possible for multiple output pipes to be bound to a single input pipe.
- *Propagate Pipes*. This type of pipe connects one output pipe to multiple input pipes. Messages are propagated from the source, output pipe, into the input pipes.
- *Secure Unicast Pipes*. They are a type of point-to-point pipes that provide a secure and reliable communication channel.

JxtaSocket and JxtaBiDiPipe

Both JxtaSocket and JxtaBiDiPipe provide bidirectional reliable and secure communication channels. Both are built on top of pipes, endpoint messengers and reliability library.

JxtaSocket and JxtaServerSocket sub-class java.net.Socket and java.net.ServerSocket respectively. JxtaSocket exposes a stream based interface, provides configurable internal buffering and message chunking. It does not implement Nagle's algorithm [13], therefore streams must be flushed as needed.

JxtaBiDiPipe and JxtaServerPipe expose a message based interface. JxtaBidiPipe does not provide chunking, applications need to ensure message size does not exceed the standard message size limitation of 64K.

Advertisements

An advertisement in JXTA is an XML document which describes any resource in the JXTA network (peers, peer groups, pipes, services, etc.). JXTA protocols use advertisements to describe and publish the existence of a peer's resources. Resources are discovered by searching for their corresponding advertisements and peers may cache discovered advertisements locally.

Each advertisement has a lifetime that specifies the availability of the associated resource. Thus, obsolete resources can be deleted without any centralized control. To extend the lifetime of a resource, an advertisement can be republished before the original advertisement expires.

Security

There are five basic security requirements that a P2P network must take into consideration: *confidentiality*, *authentication*, *authorization*, *data integrity*, *refutability*. *Confidentiality* guarantees that the message content is not disclosed to unauthorized individuals. *Authentication* confirms the identity of the sender. *Authorization* guarantees that the sender has the right to send a message. *Data integrity* guarantees that the message

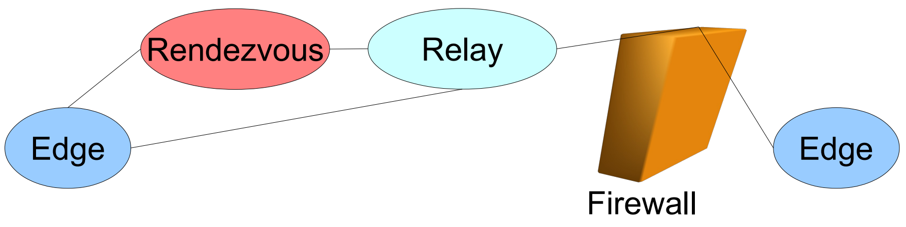


Illustration 8: Types of peers in JXTA network.

was not modified accidentally or deliberately in transit. *Refutability* guarantees that the message was transmitted by a properly identified sender and is not a reply of a previously transmitted message.

JXTA meets these requirements by adding meta-data content (credentials, certificates, digests, public keys) to XML messages. Message digests and signatures guarantee guarantee the data integrity of a message. To provide confidentiality and refutability messages may be encrypted and signed. In order to ensure message authentication and authorization, credentials must be offered.

IDs

A JXTA ID uniquely identifies a resource and is presented as a text. There are six types of JXTA entities that have JXTA ID types defined: peers, peer groups, pipes, content, module classes and module specification.

JXTA IDs use URNs in their description. URNs are similar to URIs and serve as resource identifiers that don't change over time and are location-independent.

Here is an example of a JXTA peer ID:

urn:jxta:uuid-59616261646162614A78746150325033F3BC76FF13C2414CBC0AB663666DA53903

And here is an example of a JXTA pipe ID:

urn:jxta:uuid-59616261646162614E504720503250338E3E786229EA460DADC1A176B69B731504

Each JXTA ID has an *ID Format* that is indicated right after the *urn:jxta:* prefix. The most common ID Formats are *uuid* and *jxta*. The *jxta* format is used for special identifiers as IDs of the World Peer Group and the Network Peer Group. The *uuid* format is used for most other IDs.

Content Management Service (CMS)

The CMS is a JXTA service that supports the sharing and retrieval of content within a peer group. The CMS manages the shared content for a local peer, and allows applications to browse and download content from remote peers. It works by creating advertisements for all shared content and searching through all content advertisements offered by peers in its peer group.

4.2. Network Architecture

“The JXTA network is an Ad-Hoc, multi-hop, and adaptive network composed of connected peers” [11]. Connections between peers are transient, as peers may join and leave the network at any time. For this reason message routing between peers is non-deterministic.

In practice, a JXTA network uses four types of peers (Illustration 8):

- *Minimal-Edge Peer*. It can send and receive messages, but does not cache advertisements nor route messages for other peers. It is suited for devices with limited resources (PDA, cell phones).
- *Full-Featured Edge Peer*. It can send and receive messages and may cache advertisements. It replies to discovery requests with information found in its cached advertisement, but it does not forward any discovery requests. In a JXTA application, most peers are typically edge peers.
- *Rendezvous Peer*. It is an infrastructure peer. It propagates messages, discovers advertisements and routes. It also stores a topology map of other infrastructure peers used for control propagation and maintenance of distributed hash table. A peer group can have more than one rendezvous peers and keeps track of its own set of rendezvous peers.
- *Relay peer*. It is an infrastructure peer. It aids peers behind firewalls or NAT with message transmitting.

JXTA offers a mechanism for efficiently propagate query requests within the network called *Shared Resource Distributed Index* (SRDI) service. When an edge peer publishes new advertisements it uses the SRDI service to push advertisement indexes to its rendezvous.

Thus, queries are propagated only between rendezvous nodes. This leads to faster delivery of queries from source to destination as the number of peers involved in the search is reduced.

Queries are propagated within the local network using broadcast or multicast methods. Queries beyond the local network are sent to the connected rendezvous which in turn tries to satisfy the query against its local cache and if its local cache doesn't contain the requested information routes the query to its connected rendezvous peers.

4.3. JXTA Protocols

JXTA defines a series of protocols (XML messages) for communication between peers. Peers use these protocols to discover each other, advertise and discover network resources, communicate and route messages.

There are six standard JXTA protocols. All the standard protocols are asynchronous and are based on query/response model.

- **Peer Discovery Protocol (PDP)**

Peers use this protocol to advertise their own resources and to discover resources from other peers. Resources can be peers, peer groups, pipes, services or any other resources that have advertisements. A resource is described and published using an advertisement. To discover distributed information, IP multicast is used in the local subnet and the rendezvous network which uses DHT outside the subnet. Issuing discovery query doesn't guarantee receiving a result .

- **Peer Information Protocol (PIP)**

Peers use this protocol (as a set of messages) to obtain status information (uptime, state, recent traffic) from other peers. The collected information then can be used for commercial and internal purposes. For example, the information can be used in commerce to determine the usage of a peer service and bill the service consumers for their use. Internal, the information can be used to monitor a node's behavior and reroute network traffic to improve overall performance.

The PIP *ping* message is sent to a peer to find out if this is available and to retrieve information about that peer. The PIP *ping* message specifies what to expect as a result, whether a full response (peer advertisement) or a simple acknowledgment (alive and uptime).

The *PeerInfo* message is used to send a message as a response to a *ping* message. It contains sender's credentials, source peer ID, target peer ID, uptime and peer advertisement.

- **Peer Resolver Protocol (PRP)**

This protocol enables peers to send generic queries to one or more peers and to receive responses to the query. Queries can be sent to specific peers within the scope of the group or propagated to all the peers in the group. This protocol is used to define and exchange any arbitrary information between peers.

The resolver query message encapsulates a resolver query request to a service running on another peer from the group. The resolver query message contains the credential of the sender, a specific service handler name, the source peer ID, a unique query ID, and the query. Each service can register a handler in the peer group's resolver service to process resolver query requests and generate replies. The resolver response message is used as a reply to a resolver query message and it contains the credential of the sender, a specific service handler name, a unique query ID, and the response. A peer may receive zero, one or more responses to a query request. There are no guarantees that the requests will reach their destination, nor are the results.

- **Peer Binding Protocol (PBP)**

This protocol is used to establish a virtual communication channel or pipe between one or more peers. A peer uses this protocol to bind two or more ends of the connection (pipe endpoints).

The PBP query message is sent by a peer pipe endpoint to find a pipe endpoint bound to the same pipe advertisement. In order to obtain the most up-to-date information from a peer, the query message may ask for information not contained in the local cache. If only a certain peer should respond to the query, then the query message must specify the optional peer ID.

Each peer bound to the pipe will reply with a PBP answer message which contains the Pipe ID, the peer where a corresponding InputPipe has been created, and a boolean value indicating whether the InputPipe exists on the specified peer.

- **Endpoint Routing Protocol (ERP)**

This protocol enables JXTA peers to send messages to remote peers without having a direct connection to the destination peer. A message will have to be routed to several peers before reaching its destination.

- **Rendezvous Protocol (RVP)**

This protocol is used by edge and rendezvous peers. Edge peers use it to resolve resources, propagate messages and advertise local resources, while rendezvous peers use it to organize themselves, share the distributed address space and propagate messages in a controlled fashion.

5. Data Replication Techniques

Data replication means storing copies of the same data at multiple peers thus improving availability and scalability. Full documents can be replicated or just chunks of documents or both. Since the same data can be found at multiple peers, availability is assured in case of peer failure. Moreover, the throughput of the system is not affected in case of a scale-out as the operations with the same data are distributed across multiple peers. However, the challenge in replication systems that allow dynamic updates of replicas is consistency.

We classify data replication techniques using three criteria: where updates take place (single-master vs. multi-master), when updates are propagated to all replicas (synchronous vs. asynchronous) and how replicas are distributed over the network (full vs. partial replication).

5.1. Single-master vs. Multi-master

The single-master model allows only one site to have full control over the replica (read and write rights) while the other sites can only have a read right over the replica. This model is also known as the master-slave approach due to the interaction of the master node with the other nodes (slaves) storing the replica. One of the advantages this model has is that it avoids concurrent updates at different sites during the process of centralizing updates. Moreover, it assures that only one site holds the latest version of the replica. The drawbacks this model has are that the master becomes a single point of failure and that there is the chance of master bottleneck. In case of master failure, no updates could be propagated in the future and this reduces data availability. The bottleneck appears when the master has to manage large number of replicated documents.

The single-master approach is depicted above in Illustration 9. The updates can be propagated through *push mode* or *pull mode*. In *push mode*, it is the master that initiates the propagation of the updates, while in the case of *pull mode*, the slave queries the master for existing updates.

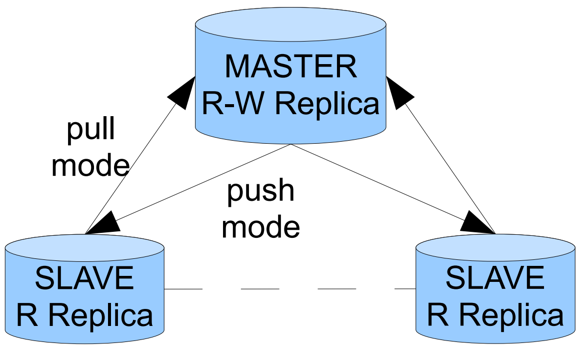


Illustration 9: Single-master replication

The multi-master model allows multiple sites the right to modify their saved replica. The system is responsible with propagating updates operated by a member of the group to all the other members and to solve any conflicts that may appear between concurrent changes made by different members. Replica reconciliation can be a cumbersome and complicated process and, in most cases, the replicas are loosely consistent. On the other hand, propagating updates from different sites in the group can lead to expensive communication. The multi-master model is more flexible than the single-master model as, in case of one master failure, other masters can modify the replicas.

The multi-master approach is presented in Illustration 10.

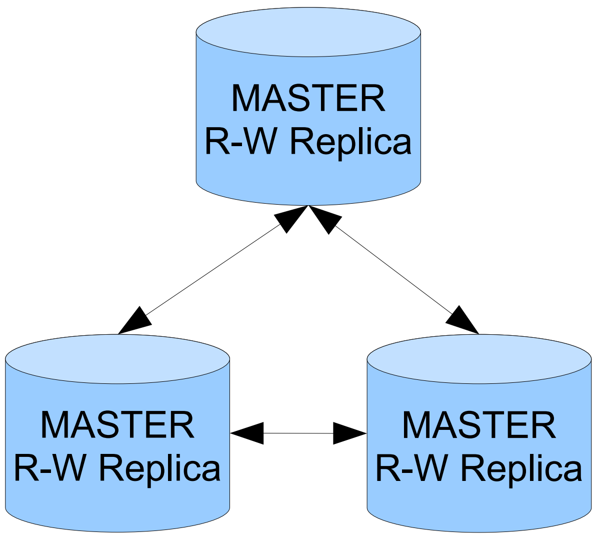


Illustration 10: Multi-master replication

5.2. Synchronous vs. Asynchronous Replication

These replication models use the notions of *transaction*, *commit* and *abort*. A transaction is a set of update operations (writes). A transaction is *committed*, that means that all the update operations that compose the transaction are performed on an object, if there are no errors. In case of errors a transaction is *aborted*. In a replicated system a transaction that updates one replica must be propagated to the other replicas in order to provide replica consistency. The update propagation can take place in two ways: the node that initiated the transaction waits for the data to have been recorded at the other nodes before finally committing the transaction (synchronous replication) or the node commits the transaction locally and afterwards propagates the changes (asynchronous replication).

5.2.1. Synchronous Replication

In this strategy (also named "eager" replication), the node that initiates the transaction propagates the update operations within the context of the transaction to all the other replicas before committing the transaction (Illustration 11). Thus, this mechanism guarantees atomic write (data is written at all sites or at none). There are several algorithms and protocols used to achieve this behavior: two-phase-locking (2PL) [2], timestamp based algorithms, two-phase-commit protocol (2PC) [2].

The above protocols bring some performance limitations – transactions that need to acquire resource locks must wait for resources to be freed if these are already taken. Kemme and Alonso [3] proposed a new protocol that makes eager replication avoid the drawbacks of the above protocols. The protocol takes advantage of the rich semantics of group communication primitives and the relaxed isolation guarantees provided by most databases. The message overhead is reduced as all writes to replicated data are postponed until the end of the transaction. Deadlocks are eliminated due to the fact that transactions are pre-ordered; this is done using group communication with which it is possible to ensure that all messages are received in the same total order at all sites. Locks are granted in the order the transactions arrive to be able to guarantee that all sites perform the same updates and in the same order. Serializability is avoided through optimizations using different levels of isolation.

But the advantage of synchronous replication of providing no divergence between replicas weights in many cases less than the disadvantages. The majority of applications wait for write transactions to complete before going further, thus the overall performance of a distributed system decreases considerably since the transaction at the source node commits only after the same transaction at the remote nodes was committed. If nodes are far way, then another problem arises: communication delays. For these reasons, synchronous replication is not recommended for systems that exceeds the order of tens of sites.

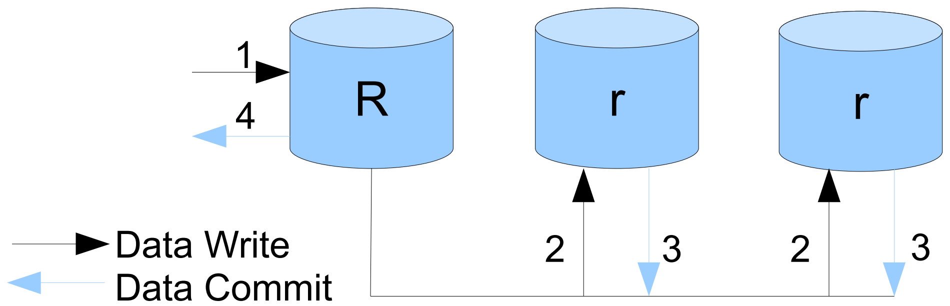


Illustration 11: Synchronous Replication

5.2.2. Asynchronous Replication

This technique does not change all replicas within the context of the transaction that initiates the updates ("lazy" replication). The transaction is first committed at the local site and after that the updates are propagated to the remote sites (Illustration 12). The asynchronous replication technique can be classified as optimistic or pessimistic in terms of conflicting updates. The optimistic approach assumes that conflicting updates will take place rarely. Updates are propagated in the background and conflicts are fixed after they happen. On the other hand, the pessimistic approach assumes that conflict updates are likely to occur and implements mechanisms for replica consistency.

With asynchronous replication, performance is greatly increased as there is no locking anymore. But if the node that initially updates the replica is down then all the other nodes will not have the up-to-date values of the replica.

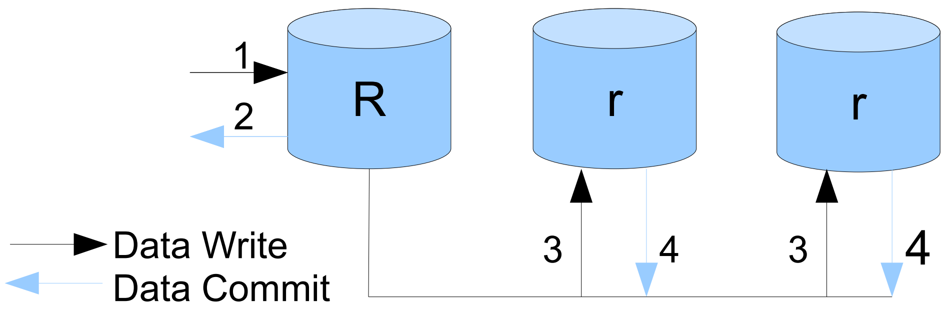


Illustration 12: Asynchronous Replication

5.2.2.1. Pessimistic approaches

These approaches combine lazy replication with one-copy-serializability. Each replicated data item is assigned a primary copy site and multiple secondary copy sites and only the primary copy can be modified. But, because primary copies are distributes across the system, serializability cannot be always guaranteed [4]. In order to solve these problems, constraints on primary and secondary copy placements must be set. The problem is solved with the help of graph representation. The Data Plecement Graph (DPG) is a graph where each node represents a site and there is a directed edge from $Site_i$ to $Site_j$ if there is at least one data item for which $Site_i$ is the primary site and $Site_j$ is the secondary site. The configurations a DPG can have for the system to be serializable is determined with the Global Serialization Graph (GSG). The GSG is obtained by taking the union of nodes and edges of the local serialization graph (LSG) at each site. The LSG is a partial order over the operations of all transactions executed at that site. A DPG is serializable only if GSG is acyclic.

The above method can be enhanced so that it allows some cyclic configurations. In order to achieve this, the network must provide FIFO reliable multicast [5]. The time needed to multicast a message from one node to any other node is not greater than *Max* and the difference between any two local clocks is not higher than *ε*. Thus, how a site receives the propagated transaction in at most *Max + ε* units of time, chronological and total orderings can be assured without coordination among sites. The approach reaches the consistency level equivalent to one-copy-serializability for normal workloads and for bursty workloads it is quite close to it. The solution was extended to work in the context of partial replication too [6].

Pessimistic approaches have the disadvantage that two replicas might not be consistent for some time interval. That is way the criterion consistency *freshness* is being used, which is defined as the distance between two replicas.

5.2.2.2. Optimistic approaches

Optimistic approaches are used for sharing data efficiently in wide-area or mobile environments. The difference between optimistic and pessimistic replication is that the first doesn't use one-copy-serializability. Pessimistic replication use synchronization during replica propagation and block other users during an update. On the other hand, optimistic replication allows data to be accessed without using synchronization, based on the assumption that conflicts will occur only rarely, if at all. Update propagation is made in the background so that it is possible to exist divergences between replicas. Conflicting updates are reconciled later.

Optimistic approaches have powerful advantages over pessimistic approaches. They improve availability; applications don't block when the local or remote site is down. This type of replication also permits a dynamic configuration of the network, peers can join or leave the network without affecting update propagation. There are techniques that allow such a thing like epidemic replication that propagates operations reliably to all replicas. Unlike pessimistic algorithms, optimistic algorithms scale to a large number of replicas as there is little synchronization among sites. New replicas can be added on the fly without changing the existing sites, examples are FTP and Usenet mirroring. Last but not least, optimistic approaches provide quick feedback as the system apply the updates Tentatively as soon as they are submitted [7].

These advantages come at a cost with system consistency. Optimistic replication encounters the challenges of diverging replicas and conflicts between concurrent updates. For these reasons, it is used only with systems in which conflicts are rare and that tolerate inconsistent data. An example are file systems in which conflicts don't happen often due to data partitioning and access arbitration that naturally happen between users.

Optimistic replication solutions can be classified by the following five parameters: operation storage, operation relationships, propagation frequency, conflict detection and resolution, and reconciliation. We describe these parameters below and then we present some existing optimistic solutions classified taking into account the above parameters [8].

Optimistic Replication Parameters

Operations. An operation is an update to an object. Operations are similar to transactions performed in the traditional databases except that they are also propagated and applied in the background, many times after they were submitted by the users. There are two ways to propagate updates taking into account the storage of operations. One way is to store operations in log files and then propagate them to remote sites to assure replica consistency. These kind of systems are called *operation-transfer systems*. Examples of such systems are *Bayou* and *IceCube*. Another way is to propagate the current state of the object. Such systems are called *state-transfer systems*. *DNS*, *Unison* and *Harmony* are included in this category.

Propagating updates to remote hosts after they had been applied at the local host incurs replica divergence. The system is only eventual consistent. This weak guarantee is not enough for some distributed systems, thus there are systems that provide stronger guarantees as a replica's state is never more than one hour old. Optimistic solutions are thus classified according to the policy for storing operations: *persistent operations* (operations are stored in log files) and *transient operations* (operations are discarded just after execution).

Operation Relationships. They represent implicit or explicit associations between operations. Conflicting updates are detected based on relationship between operations and are then arranged in a convenient order. There are four types of operation relationships that are relevant for optimistic replication systems: *happens-before*, *concurrency*, *explicit constraint* and *implicit constraint*. The *happens-before* relation is defined as the least strict partial order on operations. Let *α* and *β* be two operations executed respectively at sites *i* and *j*. Operation *α* happens-before *β* when *i* = *j* then α was submitted before *β;* or *i* ≠ *j* and *β* is submitted after *j* has received and executed *α*. C*oncurrency* takes place when neither *α* nor *β* happens before the other. An *explicit constraint* is used to dynamically represent the application semantics while *implicit constraints* are used to statically represent the application semantics.

Propagation Frequency. Propagation happens when operations and replica states are send to remote sites in order to achieve the consistency of the replica. The frequency of propagation is determined by the strategies *pulling*, *pushing* or *hybrid*. Pull-based systems update their local copies by pulling other remote sites either on demand (e.g. CVS) or periodically (e.g. DNS). In push-based systems sites send their updates to the other sites as soon as possible (e.g. LOCUS). Hybrid systems combine the first two strategies (e.g. TSAE). In pull-based, replicas arrive faster to a consistent state.

Conflict Detection and Resolution. When there is no site-coordination multiple users can update the same replica at the same time thus leading to possible conflicts. Conflicts must be detected and then eliminated using resolution approaches. Conflict detection policies are classified as *none*, *concurrency-based* and *semantic-based*. With none policy conflicts are ignored. For example, a flight-booking system could handle the same concurrent request for the same flight by picking one arbitrarily and discarding the other. But this policy leads to lost updates. Systems with concurrency-time policy declare a conflict between two operations based on timing of operation submission (e.g. LOCUS). Systems that are aware of operations' semantics can reduce conflicts (e.g. Bayou, IceCube). In the above flight-booking system, two concurrent reservation requests for the same flight are permitted as long as the flight dates are not the same. Conflict resolution is highly application specific and can be either *manual* or *automatic*. Manual approach is adopted by the majority of the systems and requires user intervention to fix it (e.g. CVS). But there are systems that solve the conflict automatically (e.g Bayou).

Reconciliation. A certain replica can be modified at different sites, thus permitting applications to continue functioning even if some sites are are offline or down. This kind of parallel updates can cause replica divergence. The process of reconciliation brings these replica to a mutual consistent state. There are different reconciliation strategies that depend on the type of input information and the criterion for ordering updates. The input information can be the current states of the replicas or the update operations. *State-based reconciler* is a reconciliation engine that takes as input the updated states of the replicas and tries to make them as similar as possible (e.g. Harmony, Unison). *Operation-based reconciler* is a reconciliation engine that accepts as input the history of operations performed at each replica and tries to build a common sequence of operations (e.g. Bayou and IceCube). The criterion for ordering updates can be based on ordinal information associated with updates or on semantic properties. The *ordinal-reconciler* tries to maintain the submission of updates based on when, where and by whom updates were performed. This is achieved using timestamp-based ordering (e.g. TSAE) or version vectors (e.g. LOCUS). The semantic-reconciler exploits semantic properties associated with updates to reduce conflicts (e.g. IceCube).

Existing Optimition Solutions

- **DNS** (Domain Name Server) is the standard hierarchical name service for the Internet. The responsibility of assigning domain names and mapping those names to IP addressees is assigned to authoritative name servers (plays the role of the master) for each domain. These in turn can assign other authoritative name servers for their sub-domains. The mappings are stored in a database. The master can maintain a list of slave servers that copy the database from the master and both master and slaves can answer queries from clients and remote servers. Only the database stored at the master is updated and when this happens the timestamp is incremented. Slaves periodically poll the master and update the database when they find a different timestamp at the master site. There is a risk that, for some period of time, the contents of a slave might be out of date and clients may notice old values.

- **LOCUS** is a distributed operating system composed of a replicated file system. A conflict takes place when there are at least two concurrent updates on the same object. Conflicts among replicas are solved using version vectors. It automatically resolves conflicts by taking two versions of the object and creating a new one.
- **TSAE** (Time-stamped anti entropy) uses real-time clocks to order operations. Ack vectors are used in conjunction with vector clocks so that each site can find out about the progress of other sites. The ack vector *AVi* on Site *i* is an *N*-element array of timestamps. $AV_i[i]$ is defined to be $\min_{j\in\{1\ldots M\}}(VC_i[j])$, i.e., Site *i* has received all operations with timestamps no newer than $AV_i[i]$, regardless of their origin. Ack vectors are exchanged among sites and updated by taking pair-wise maxima, just like VCs. Thus, if $AV_i[k] = t$, then *i* knows that *k* has received all messages up to *t*. Thus, all operations with timestamps no larger than $\min_{j\in\{1\ldots N\}}(AV_i[j])$ are guaranteed to have been received by all sites, and they can safely be executed in the timestamp order and deleted. TSAE does not perform any conflict detection or resolution [7].
- **Ramsey and Csirmaz's file system** with support for few operation types, including create, remove and edit. For every possible pairs of concurrent operations, they define a rule that specifies how the operations interact and may be ordered. Non-concurrent operations are applied in their happens-before order. For example, the creation of two files in the same directory is allows. Or, if one user modifies a file and another deletes its parent directory, it marks them as conflicting and ask the user to repair them manually.
- **Unison** is a file syncronizer. It uses the states of the replicas to reconcile two replicas of a file or directory. Unison takes into account the semantics of the file system when trying to merge two replicas. Only non-conflicting updates are propagated. Thus, it is possible that replicas may hold different states after reconciliation.
- **CVS** (Concurrent Version File System) is a version control system that lets users edit a group of files collaboratively and retrieve old versions on demand. There is one single site that holds the authoritative copies of the files along with all changes committed to them in the past. Users create private copies of the files and edit them

using standard tools. When users commit its private copies to the repository, CVS automatically merges changes if there is no overlap. Otherwise, each user has to solve conflicts manually.

- **Harmony** synchronizes disconnected updates to replicated tree-structured data. It is used to reconcile the bookmarks of multiple web browsers (Mozilla, Safari, OmniWeb, Internet Explorer and Camino). This application allows bookmarks and bookmark folders to be added, deleted, edited and reorganized by different users in disconnected machines. Harmony takes only replica states and it does nor resolve update conflicts.
- **Bayou** is a research mobile database system. It lets a user replicate a database on a mobile computer, modify it while disconnected and synchronize with any other replica of the database that the user happens to find. Update operations (SQL statements) operated at one site are propagated to other sites epidemically. A site applies operations tentatively as soon as they are received from the client or the other sites. There are chances that sites receive operations in different orders. If this is the case, sites must undo and redo operations repeatedly as they gradually learn the final order. Each operation has attached a *dependency check* and a *merge procedure*. Conflicts are detected using *dependency checks* and resolved using the *merge procedure*. A designated master site takes the final decision regarding ordering and conflict resolution. The master orders operations and resolves conflicts in the order of arrival and sends the decisions to other sites epidemically as a side effect of ordinary operation propagation. Bayou is a multi-master, operation-transfer system that uses epidemic propagation over arbitrary, changing communication topologies.
- **IceCube** supports multiple applications and data types using constraints between operations (actions). The update operations are stored in logs. Constraints can be supplied from several sources: the user, the application, a data type, or the system. Reconciliation here is an optimization problem where the goal is to find the largest set of actions that do not violate the stated constraints.

5.3. Full vs. Partial Replication

Placing a replica over the network has an impact on replica control mechanisms. This section presents the two main approaches for replica placement: full replication and partial replication.

Full replication takes place when each participating site stores a copy of every shared object. This requires each site to have the same memory capacities and maximal availability as any site can replace any other site in case of failure. Illustration 13 shows how two objects A and B respectively are replicated over three sites.

In the case of *partial replication*, each site holds a copy of a subset of shared objects so that sites can keep different replica objects (Illustration 14). This type of replication requires less storage space and reduces the number of messages needed to update replicas since updates are propagated only to the sites that hold a certain replica. Thus, updates produce reduce load for the network and sites. But, the propagation protocol becomes more complex since the replica placement must be taken into account. Moreover, this approach limits load balance possibilities since certain sites are not able to execute a particular type of transactions [7]. In partial replication, it is important to find a right replication factor (the number of documents out of the total number of documents each peer stores). Careful planning should be done when deciding which documents to replicate and at which peers.

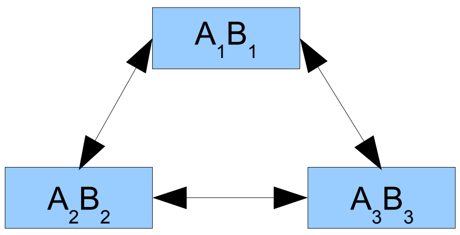


Illustration 13: Full replication with two objects A and B.

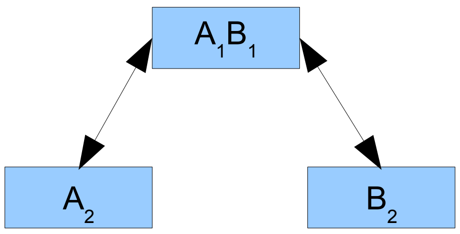


Illustration 14: Partial replication with two objects A and B.

6. System Architecture

This chapter presents the architecture of the replication system. It focuses on peer-group design, super-peer and peer structures. The system is built using JXTA library.

6.1. Peer-group Design

A peer-group is a gathering of multiple peers who can be geographically far away from one another but who have a common goal. For example, they could all work together accomplishing a group-project (Section 6.1.1.).

The peer-group is organized in such a manner that each peer can directly communicate with any other peer in the group (Illustration 15). Furthermore, the peer-group has a central manager, the super-peer, who has two important roles. One role regards only the peer-group in which the super-peer resides. The super-peer keeps track of the project development and assigns tasks to the peers in the group. The other role regards the whole network. The super-peer facilitates the communication with other peers in other peer-groups.

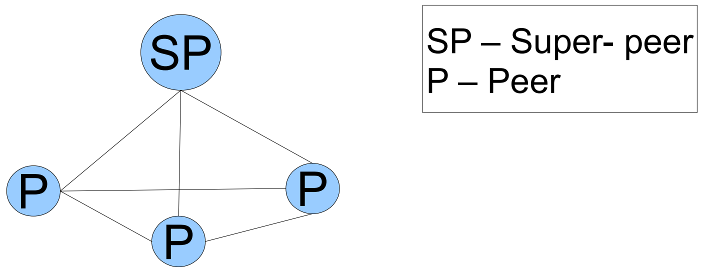


Illustration 15: Simple Peer-group Structure

The simple peer-group structure is enhanced with a Data Repository (DR) that acts as a storage facility in order to keep the initial documents related to the project (Illustration 16). When a peer receives a task, it then connects to the DR to get copies of the documents which it can modify afterwards.

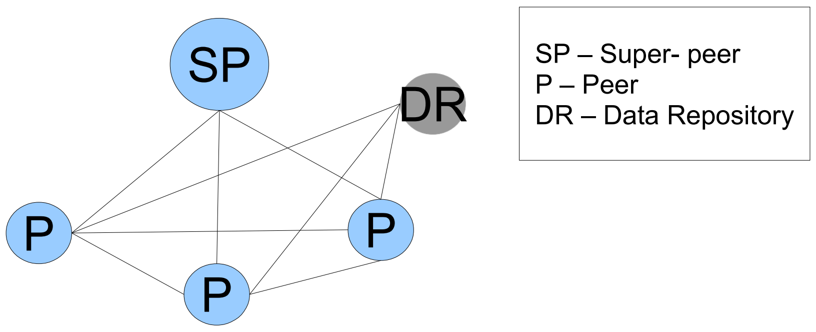


Illustration 16: Peer-Group Structure with a Data Repository

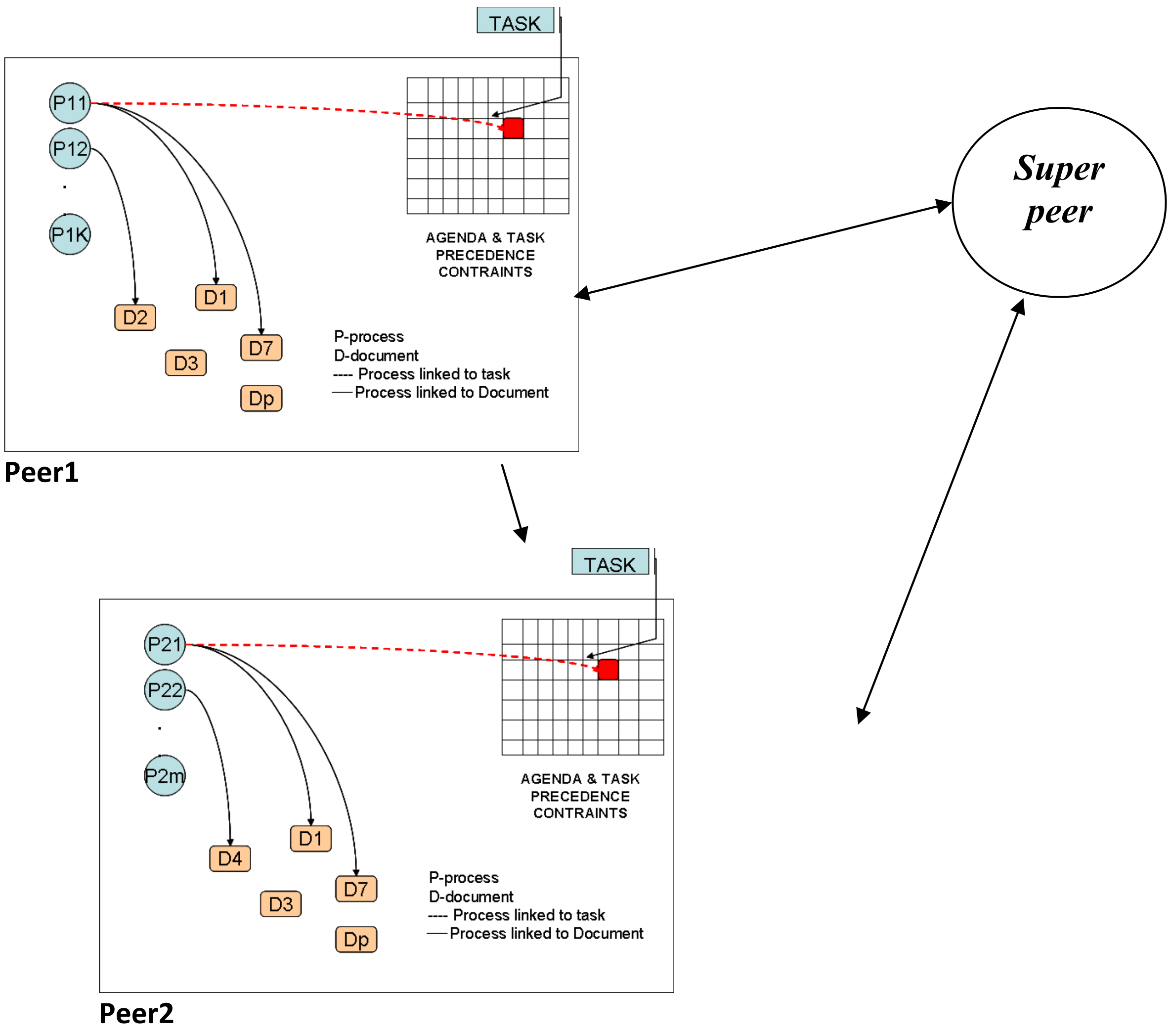
In JXTA, a peer-group is uniquely identified by a `PeerGroupID` (see Section 4.1.). We design our peer-group with a custom membership policy that requires each peer credentials (user and password) when joining the group. The communication within the members of the group is done using unidirectional communication and multicast.

6.1.1. The Group-Project

A group-project consists of several tasks that are to be completed by the peers in the group. Each task implies document replication, document updates and convergence of replicas among several peers from the group. The description of the group-project is a table similar to the one below:

Task			Assigned Peers	Start date	End date	Task Precedence (dependencies)	Documents (associated to task)
Id	Name	Description					
7	Revision	Revision of the design document	P1, P2	date1	date2	Requires completion of tasks Id 4, 6	D1, D3

Documents are replicated among the peers of the group in order to ensure availability. But documents might suffer modifications while stored at different peers. In order to keep the replicas consistent, a mechanism for convergence of the replicas is needed. We assume that each peer modifies only its “own” documents and update replicas of other documents locally. We have chosen this strategy in order to avoid concurrent updates that would have led to either locking the document while modifying it or applying complex conflict detection and resolution policies.


TASK
P11
P12
P1K
D2
D1
D7
D3
Dp
AGENDA & TASK PRECEDENCE CONTRAINTS
P-process
D-document
---- Process linked to task
— Process linked to Document
Peer1
Super peer
TASK
P21
P22
P2m
D4
D1
D7
D3
Dp
AGENDA & TASK PRECEDENCE CONTRAINTS
P-process
D-document
---- Process linked to task
— Process linked to Document
Peer2

6.2. Super-peer

A super-peer acts as a server for some peers, namely the peers found in the same group with it, and as an equal in a network of super-peers (Illustration 17). The advantages of having a super-peer is that job submissions and data queries arrive faster to the destination. The super-peer in our peer-group structure plays the role of a rendezvous peer: maintains global advertisement indexes and assists edge peers with advertisement searches. This means that the super-peer is aware about the other super-peers in the network and thus it makes the connection between the peers in the group and the other peers and super-peers from the network.

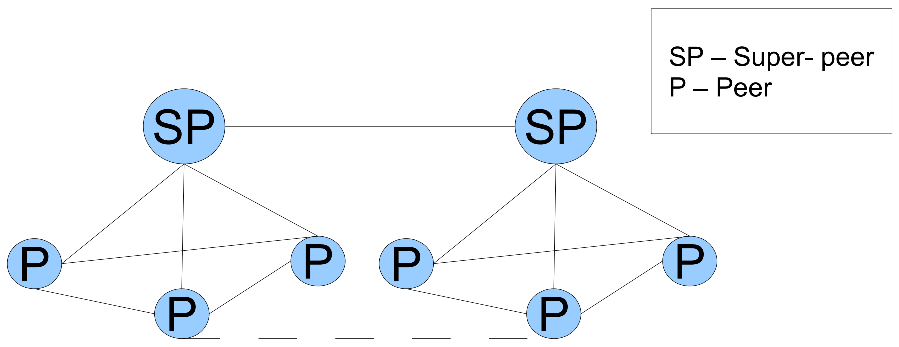


Illustration 17: Super-peer network

The super-peer architecture is presented in Illustration 18:

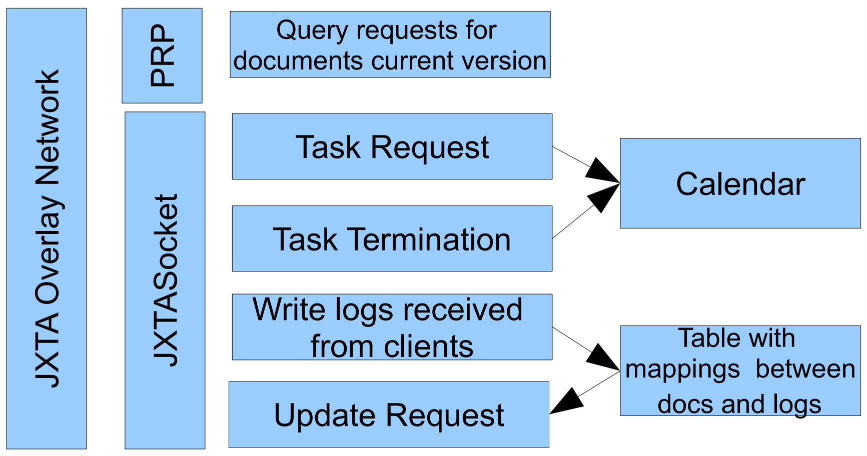

JXTA Overlay Network
PRP
JXTASocket
Query requests for documents current version
Task Request
Task Termination
Calendar
Write logs received from clients
Update Request
Table with mappings between docs and logs

Illustration 18: Super-Peer Architecture

The *Calendar* structure keeps track of the progress of the project: tasks that have been submitted to peers, which peers finished a certain task, what tasks are finished by all peers, what tasks are in stand by, etc.

The super-peer receives every update operation from online peers. An update operation consists of several write operations. We call the set of write operations a write log (Illustration 19). The super-peer keeps in a table the mappings between document IDs and the corresponding write logs. This table is structured in the form of a hash table, where the keys are IDs of the documents that are replicated among the peers of the group and and the values are vectors of write logs, sorted by the version of the update. The super-peer can thus respond to update request issued by late joining peers.

```
/document/part[1] soare
/document/part[2] ABC
/document/part[3] yupii
```

Illustration 19: Example of a write log

In order to avoid storing thousands of updates at the super-peer and thus risking the super-peer to run out of memory, it is necessary that the super-peer queries periodically the peers in the group asking for their last versions of the documents. Thus, the super-peer uses the Peer Resolver Protocol (section 4. 3) in order to send queries requesting the last version of the documents stored at the peer nodes. The super-peer will then delete part of its logs only if all the peers in the group transmitted their last version of the documents.

Most part of the communication between super-peer and peers is realized through *JXTASocket.* There is one *JXTAServerSocket* that listens for incoming connections. Each time a connection is accepted, a thread is started for that connection. In this way, the super-peer can handle multiple connections simultaneously.

The super-peer accepts the following types of connections from the peers: for requesting a task, for task termination, for transmitting changes and for requesting updates. The threads created for *Task Request* and *Task Termination* operate modifications in the task entry from the calendar object. The thread created for *Write logs received from clients* adds the received log with updates to the *Table with mappings between docs and logs*. The thread created for *Update Request* retrieves logs with versions greater than the ones stored at the requesting peer and sends those logs to the peer.

When a task finishes, all replicas are in a final state.

6.3. Peer

A peer is responsible for solving a certain task. After the peer receives a task from the super-peer, it then contacts the DR to download the files related to the received task. Then, the peer will operate changes only on the documents that has the right to modify. Each commit operation is first send to the super-peer through socket communication and then propagated through multicast among the members of the group. The super-peer is the first who receives the changes because the multicast protocol is unreliable and thus there exist the danger that some peers will not receive the commit updates through multicast. In this case, the peers can still acquire the latest updates from the super-peer. Each peer keeps an integer representing the current version of the document it stores at his site. This current version number is actually a counter that increases every time the document is updated.

The peer architecture that is related with the interaction with the super-peer is presented in Illustration 19. The peer communicates with the super-peer through sockets. This is because sockets provide reliable transmission of data and there is no need for supplementary verification at peer site. The peer sends task requests, task termination notifications, commit changes and update requests to the super-peer (the peer is in push mode). The peer is also responsible with transmitting the current version of documents in response to query requests asking for it sent by the super-peer using JXTA Peer Resolver Protocol.

The peer architecture that is related with the peers in the group is presented in Illustration 20. The peer uses multicast to propagate changes (update operations) to the other peers. The peer propagates with the change the current version of the document too.

The peer architecture that is related to the Data Repository (DR) is presented in illustration 21. The peer uses the *Content Management Service* offered by JXTA to download files from a specified location (see Section 4.1.).

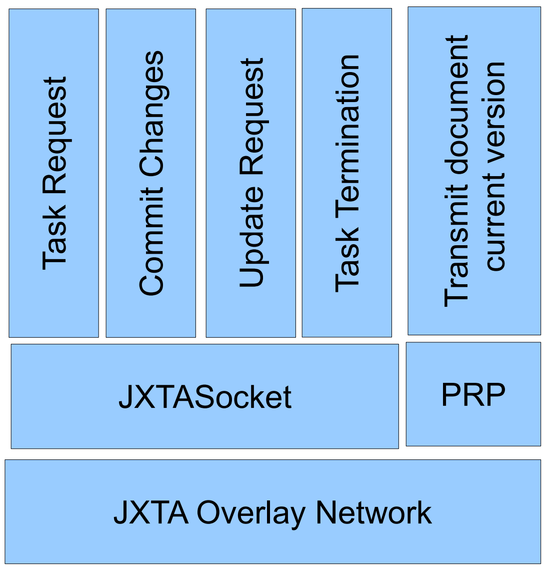


Illustration 20: Peer Architecture: interaction with the super-peer

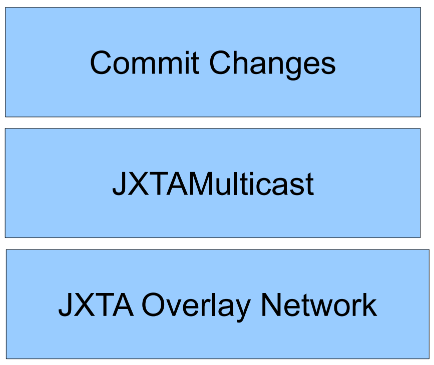


Illustration 22: Peer Architecture: interaction with peers in the group

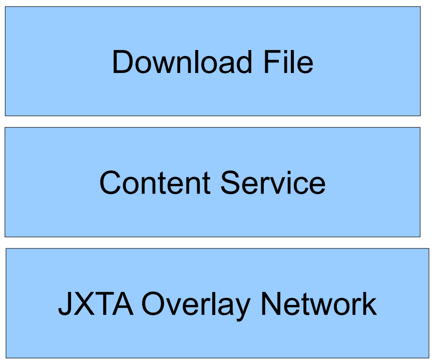


Illustration 21: Peer Architecture: Interaction with the Data Repository

7. System Implementation

In this chapter we detail the protocols implemented for the communication between peers and between peers and super-peer as well as the mechanism through which peers acquire the documents that they will change later during task solving.

7.1. Communication Protocols

We present the protocols implemented for the communication between two peers and between a peer and the super-peer. These protocols are used in implementing the architecture described in the subsections 6.2. and 6.3.

7.1.1. Peer – Super-peer Protocols

The communication between a peer and a super-peer uses the following protocols: *Task Request*, *Commit Changes*, *Update Request* and *Task Termination*. These protocols are represented in the form of message sent through sockets. They are initiated by peers. In the case of the protocols *Commit Changes* and *Task Termination* the super peer is not supposed to reply. Each message sent by the peer to the super-peer is differentiated by the first byte of the message, which represents the code of the message.

Task Request

This protocol is used whenever a peer connects to the super-peer for requesting a task (the peer is said to be in pull mode). The peer usually requests tasks when it enters the network for the first time or after it finishes the current task and asks for new tasks to be assigned to it. The super-peer must answer in reply to the request received. It will send a new task if there are still tasks to be solved or signals the fact that there are no more unsolved tasks left.

The message format used by the peer to send the Task Request to the super-peer is:

<*TR_Code PeerID*>

TR_Code – Task Request code. Has the value “0”. It occupies one byte.

PeerID – the ID of the peer that made the request. It is an integer and thus occupies 4 bytes. The super-peer must be aware of which peer sends the request, as a certain task might not be solved by all peers in the group.

The super-peer will reply using the following message format:

<*Task_ID P list_of_peers D list_of_docs R list_of_replicas*>

Task_ID – the id of the task to be solved.

P – indicator signaling that it follows the list of peers involved in this task.

list_of_peers – list of peers IDs involved in the task. The peers IDs are separated through spaces.

D - indicator signaling that it follows the list of documents associated with the task. The list of documents contains a list of documents that can be modified by this peer and a list of replicas that cannot be modified by this peer, but by some other peer in the group.

list_of_docs – list of IDs of documents that can be modified by the current peer. The IDs are separated through space.

R – indicator signaling that it follows the list of replicas

list_of_replicas – list of IDs of documents that cannot be modified by the current peer. These documents are modified by collaborative peers. This peer will receive updates for these documents. The IDs are separated through space.

Commit Changes

This protocol is used to send the super-peer the changes of a document.

The message format used by the peer to send the changes to the super-peer is:

<*C_Code DocID version update*>

C_Code – The code of the operation *Commit Changes*. Has the value “1”.

DocID – the ID of the document that has been updated by the source peer. The ID is a string because we also use the ID as the name of the document.

version – the current version of the document

update – it is a set of writes that were operated on the document. Each write consists of the node of the document that is going to be modified and of the new value.

The super-peer is not supposed to reply to this message.

Update Request

This protocol is used in the moment a peer changes the status state from *OFFLINE* to *ONLINE* and through which the peer sends an update request to the super-peer. The update request contains the current versions of the documents that the peer already has.

The message format used by the peer to request from the super-peer updates that were made while the peer was offline is:

<UR_Code listDocVersion>

UR_Code – Update Request code. Has the value "2".

listDocVersion – list with versions of documents. It is a list of pairs DocID:Version, separated by spaces.

The super-peer responds with

The super-peer will reply using the following message format:

<(*docID version*
updates)*>

docID – document ID

version – the version number of the document

updates – log of writes. Each write is a pair node-value, where node is the document node that was modified and value is the new value of the node.

Task Termination

This protocol is used whenever a peer finishes the task that has been attributed. The super-peer must know when a task finishes in order to be able to assign new tasks for the accomplishment of the project.

The message format used by the peer to signal the termination of a task is:

<*TT_Code TaskID PeerID*>

TT_Code – Task Termination code. It occupies one byte and has the value "3".

TaskID – ID of the task that has been finished by the current peer.

PeerID – ID of the peer that finished the task.

7.1.2. Peer – Peer Protocols

A peer communicates with other peers only when the first modifies a document and needs to commit changes to the collaborative peers. The protocol is *Commit Changes* (subsection 6.2) and implies multicast communication.

The message format is:

<*docID version updates*>

docID – ID of the document modified

version – the current version of the document

updates – log of writes. Each write is a pair node-value, where node is the document node that was modified and value is the new value of the node.

7.2. Super-peer cache

Super-peer stores all the write logs that the peers from the group are sending through multicast. In time, the size of the cached logs can become very big and thus, in order to make space for the new logs, the old ones must be deleted. There are two alternatives. One is to assign a time to live (TTL) field to each write log and erase those logs whose TTL field reaches zero. In this case, there are peers that will loose some updates after rejoining the network. The second alternative is for the super-peer to periodically send queries requesting for the current versions of the documents stored by the peers in the group. If all the peers answer to the query, then the super-peer deletes from its cache all the updates already committed by all the peers.

7.3. Document Representation

We choose to use XML documents as document replicas. They are easy to manipulate using XML database systems. The documents we have used in the replication system were built using the following document data type definition (DTD) schema:

We use the following DTD associated with the documents:

```
<!ELEMENT document (part+ )>
```

```
        <!ATTLIST document id CDATA #REQUIRED>
    <!ELEMENT part (#PCDATA)>
```

We have assumed this sufficiently fine fine-grained description so that any change can be automatically done to the replicas. This XML structure is transparent to replication techniques and fully depends on user design.

In order to easy modify the content of the XML documents we found it necessary to use a XML Database. We have chosen the *BaseX XML* Database because it is free, can be integrated with Java and has XQuery API [12].

Update operations on XML files mean changing the value of a certain node. For that, we have used the following Xquery syntax:

```
replace value of node /document/part[1] with new_value
```

We have used the following class for updating documents:

```
class DocUpdater:
  DocUpdater(String docID)
   updateNode(String node, String new_value) //updates the node node with the value
                                              //new_value
  writeToFile()//writes the content of the XML database to disk
 query(final String query) //executes an Xquery on the current document
```

8. Experimental Study

We want to study how much time does it take for replicated documents to get in a consistent state using multicast for update propagation in a cluster environment. Moreover, we wanted to study how does the content of a write log affect the performance of pushing received updates at client side and how our algorithm for reducing the super-peer cache size manages updates drop off.

We conducted our tests using the Computing Cluster System from the Llenguatges and Sistemes Informàtics Department (LSI) at the Universitat Politècnica de Catalunya (UPC). The LSI cluster is based on queues: Oracle Grid Engine (SGE). The elements of the cluster (execution nodes, servers and login hosts) are linked to a gigabit switch (2x48 ports), so they are part of a private LAN. Every single host has a link speed of 1 Gb full-duplex.

To evaluate the performance of our replicated system described in Section 7, we simulated different scenarios for the asynchronous collaboration of groups with one super-peer and the number of peers ranging from two to six. The scenarios differ by the the number of replicas in the system or by the number of peers or by the time peers are offline. A certain document is supposed to be updated by a single peer which is responsible with propagating the changes to the other peers.

Document consistency

The average time for a document to achieve consistency is a sum between the time needed to multicast the updates (network time) and the time needed to operate the changes in the local database (client time).

The time needed to multicast updates strongly depends on the overlay network, more exactly on the number of routers used to replicate the package so that it arrives to the destination peer and on the performance of deployed switches. *Table 1* shows the average time (network time) needed to propagate changes through multicast of a single document of size 5KB to groups formed by 2 peers to 6 peers.   Nodes in the cluster are connected through switches.

Table 1: Propagating updates for a single document

#Peers in the group	2	3	4	5	6
Time (ms)	6	6	6	6	6

The time needed to operate the changes in the local database depends not on the size of the write log, but on the number of operations in the write log. *Table 2* shows the time in milliseconds needed to perform an update to the local database taking into account the number of operations and the size of the write log. It can be seen that executing just one operation that updates one field with a new content of size 600 bytes takes very little time comparing to executing 100 operations for updating fields with new content of size 6 bytes each or to executing 200 operations for updating fields with new content of size 3 bytes each.

Table 2: Time (ms) needed to update the local database

	Size of write log (bytes)	
# operations	**600**	**1200**
1	3	4
100	466	517
200	671	683

Late join data delivery

When a peer joins the network, the first thing it does is to update its current state. Thus the data transmission is initiated by the peer itself when sends an update message to the super-peer. We chose this solution in order not to cache operations at peers sites and thus polling each peer in the group which would have caused too much traffic and the network could have got saturated. Another advantage is that the newly joined peer obtains the updates fast, as it only has to contact a single node (the super-peer) which already has all the updates.

The update message encapsulates the IDs of the documents for which the peer wants to retrieve updates and the current versions of the documents. The super-peer is responsible with delivering the updates the peer requested as soon as possible. For this reason, the super peer uses a hash table which maps document IDs with a vector of write logs. The write logs for a certain document ID are inserted into a vector which is sorted by the version of the update. The super-peer then retrieves from its cache with a complexity of O(1) the updates to be transmitted to the peer.

Super-peer cache size

In our super-peer network, the super-peer is responsible with caching all the updates operated by all the peers in the group. Thus, the super-peer requires a big storage capacity. But even in this case, the upper limit can be reached. That's why there is a need for old cached updates to be dropped.

We analyzed two techniques to erase old updates. One technique implies setting a time to live field to each received update. The other assumes the super-peer to send queries periodically requesting the current version of documents stored at each peer and then erase all the updates that had been operated by all the peers. The first technique has the drawback to erase updates that were not yet enforced by some peers in the group while the other causes network traffic and for the super-peer the obligation to store many updates for peers that are offline most of the time and thus does not respond to such queries.

In order to evaluate the above mentioned techniques, we chose to simulate the behavior of 3 peers, each peer sending through multicast when it is online one update per second during a time span of 300 seconds (Illustration 23). Each peer is online 80% of the simulation time. Illustration 20 shows the online time of peers. For the first technique, the super-peer assigns to the TTL field the value of 30 seconds and, for the second technique, sends queries every 30 seconds requesting the current version of documents.

Evaluation of these two techniques can be seen in Illustration 24. Using the second technique, the number of operations cached at super-peer is significantly greater than in the first technique, though the second technique makes possible for the super-peer to empty its cache when all the peers are online and answer to the queries with their current version of the documents. The biggest problem with deleting cached updates when TTL reaches zero is that peers will not receive certain updates when they rejoin the group. For example, at simulation time 130s, peer 2 is back online, but will not receive 60 operations, 30 operations from peer 1 and 30 operations from peer 3.

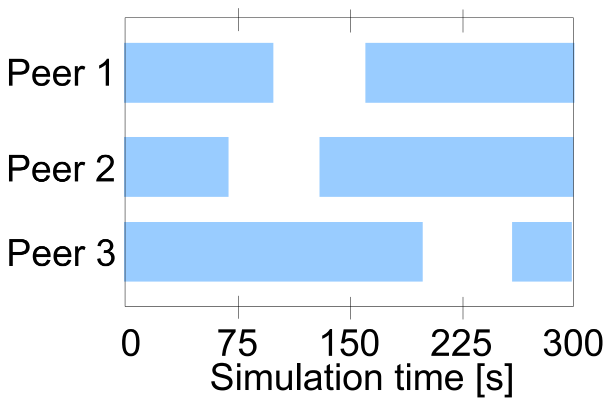


Illustration 23: Online time of peers

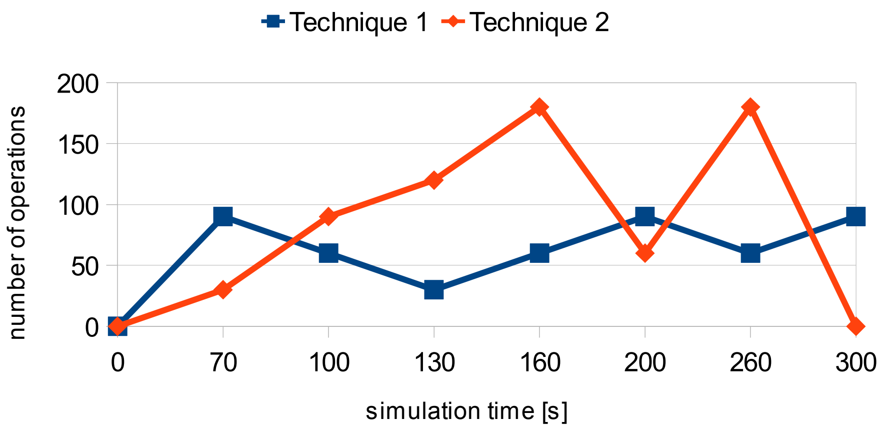


Illustration 24: Number of operations cached at super-peer

9. Conclusions

We have designed a replication system, having a super-peer network and using optimistic replication techniques. In this system, replicas can be dynamically updated and the updates are propagates through multicast to all the members of the group thus ensuring a fast update delivery. The time needed to deliver the updates depends especially on the underlying network topology. As multicast packages are replicated by the routers in the network, the number of routers on the path from source to destination contributes in a great measure to the propagation time of the updates. Moreover, the performance of the switches has also an important role in multicast scalability.

Our system permits both full and partial replication. The peers are responsible with filtering the updates that aren't interested in. The performance of the network doesn't change in this case, only peers require less storage space and they don't use their computing resources for updating all the objects existing in the system.

Though multicast ensures a fast delivery of content to all members of the group, the datagram packet is limited in size (the maximum IP packet size is 65535). Thus, write logs exceeding this size must be split up and mechanisms for packet ordering at the destination must be used.

Multicast performance and scalability depends on the underlay network, thus the consistency time in replication system strongly depends on data communication.

BIBLIOGRAPHY

[1] Pasquale Cozza, Domenico Talia, “A Super-PeerModel for Multiple Job Submission on a Grid”, CoreGRID Technical Report Number TR-0067, 2007

[2] Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman, “Concurrency Control and Recovery in Database Systems”, 1987

[3] Bettina Kemme, Gustavo Alonso, “A new approach to developing and implementing eager database replication protocols”, ACM Transactions on databse systems, 2000

[4] P. Chundi, D. Rosenkranz, “Deferred Updates and Data Placement in Distributed Databases”, 1996

[5] E. Pacitti, P. Minet and E. Simon, “Fast algorithms for maintaining replica consistency in lazy master replicated databases”, 1999

[6] C. Coulon, E. Pacitti and P. Valduriez, “Consistency management for partial replication in a high performance database cluster”, 2005

[7] Yasushi Saito, Marc Shapiro,“Optimistic Replication”, ACM Computing Surveys, 2005

[8] Vidal Martins, Esther Pacitti, Patrick Valduriez, “Survey of data replication in P2P systems”, 2006

[9] R. Huebsch et al., “Querying the Internet with PIER”, 2003

[10] Beverly Yang, Hector Garcia-Molina, “Designing a Super-Peer Network”, Proceedings of the 19th International Conference on Data Engineering, 2003

[11] JXTA Java Standard Edition v2.5: Programmers Guide

[12] http://basex.org/

[13] John Nagle, “Congestion Control in IP/TCP Internetworks”, 1984

Project Costs

In this section we estimate the cost this project would have, taking into account the hardware infrastructure (Table 3), the software applications (Table 4) and number of working hours (Table 5).

Table 3: Hardware costs

Product	Type of the Product	Price
Pentium D Dual Core 2.8 GHz, 4 Gb Ram, 120 Gb HD	From the enterprise	1 300 €
Intel Core Duo CPU 2GHz, 2GB RAM x 10	From the enterprise	3000€
Total		4 300€

Table 4: Software costs

Product	Type of Product	Price
Eclipse 3.5	OpenSource	0€
BaseX	OpenSource	0€
OpenOffice	OpenOffice	0€
Total		0€

Table 5: Personal costs

Type of work	# hours
Learning	100
Analysis	60
Design	110
Implementation	90
Testing	60
Documentation	120
Total	340h

The price of a working hour is 8.5€, thus the personal revenue is 2890€.

The total cost of the project consists of summing the previous mentioned costs:

Table 6: Total cost of the project

	Costs
Hardware	4300€
Software	0€
Personal	2890€
Total	7190€

Project Plan

TASK	START	END	DESCRIPTION	DELIVERY	DELAY
State of the art	02/15/11	03/06/11	Literature review, system infrastructure	report	none
JXTA Technology	03/07/11	03/20/11	Familiarizing with JXTA protocols, JXTA API		none
Architecture	03/21/11	04/11/11	Building the system architecture	report	none
Implementation	04/12/11	05/01/11	Implementing the replication system	code	none
Documentation	05/01/11	05/30/11	Write documentation	paper	none
Experimental Study	06/01/11	06/14/11	Testing on cluster	report	+ 3 days