Diploma Thesis

iServerP2P

Distributed iServer Architecture Based on Peer-to-Peer Concepts

Christian Heinzer, D-INFK
heinzerc@student.ethz.ch

February 26th 2004

Institute of Information Systems
Swiss Federal Institute of Technology (ETHZ)

Diploma Professor:
Prof. Moira C. Norrie

Supervisor:
Beat Signer

Abstract

The Integration Server is a framework that provides management of links amongst mixed-media content. Plug-ins allow to integrate any type of media. This work extends the existing architecture with the possiblity to exchange link data in a decentralised fashion with other Integration Server instances. To achieve this, the JXTA Java peer-to-peer framework is used. The data is exchanged in XML format. Plug-ins implement the transformation from the media referenced by a link to its XML representation and reverse. Desired links are searched by broadcasting queries, the arriving answers are small, specifying all matching links. Those matches are rated and in a second step the best links can be requested. As user management is an important part of Integration Server, a basic update of user data over peer-to-peer is implemented.

Contents

1 Introduction **1**

2 Peer-to-Peer **3**
2.1 Definition . . . 3
2.2 Architectures . . . 3
2.3 Bootstrap Problem . . . 4
2.4 Application Range . . . 5
2.5 Example Applications . . . 6
2.5.1 P2P Telephony: Speakfreely . . . 6
2.5.2 P2P Radio: Streamer . . . 8
2.5.3 Pure P2P search: Gnutella . . . 8
2.6 iServerP2P Requirements . . . 9

3 JXTA **11**
3.1 Technical Terms . . . 11
3.2 History and Principles . . . 13
3.3 Available Extensions and Applications . . . 13
3.4 Comparison of Gnutella and JXTA . . . 14
3.5 Protocol Stack . . . 15

4 JXTA in use **17**
4.1 Groups . . . 17
4.2 Resolver . . . 18
4.3 Pipe . . . 19

5 User Management **21**

6 Link Exchange **25**
6.1 Queries . . . 25
6.2 Hash Identification . . . 26
6.3 Resolver: Query ID . . . 27
6.4 Pipe Request . . . 28
6.5 Synchronization . . . 28

7 Link Rating **29**
7.1 Link Rater Examples . . . 31

8 Serialization and Materialization **33**
8.1 OM Types . . . 33
8.2 Methods for Users . . . 35

8.3 Link Specific Issues . . . 37
8.4 Transient and Persistent Objects . . . 38
8.5 Plug-ins Explained . . . 39

9 Manual for Users and Developers 41
9.1 Configuration . . . 41
9.2 Startup . . . 41
9.3 Queries and Responses . . . 42
9.4 Basic Example . . . 42
9.5 Application Integration . . . 42
9.6 Troubleshooting . . . 42
9.6.1 JXTA Startup Problems . . . 43
9.6.2 java.io.IOException: Negative Seek Offset . . . 43
9.6.3 OutputPipeListener . . . 43

10 Outlook and Alternatives 45
10.1 JXTA 2.2 + . . . 45
10.2 Services . . . 45
10.3 Pipes . . . 45
10.4 Link Identification . . . 45
10.5 Link Rating . . . 46

11 Personal conclusion 49

Acknowledgments 50

References 51

A Task Description 53

B API Reference 54

1 Introduction

The Integration Server [ISERVER, url] supports the management of link data (general association concept between arbitrary types of media) and its storage in a database.

The current implementation is based on OMSJava [OMSJAVA, url], an implementation of the OM model [NORRIE, 93].

There exists an iServer application (Paper^{++} plug-in) that offers predefined data types and structures. As part of the Paper^{++} project, a tool to simplify authoring of link data has been developed. [NORRIE, 03]

The goal of this thesis is to extend the existing iServer framework with decentralised access to other users' link data so that link knowledge can easily be shared.

We do not want to rely on a central server for organisation of communication or even data storage for reasons like scalability, speed, flexibility, administrative and hardware costs or breakdown security.
The resulting extension is called iServerP2P for short.
iServerP2P challenges and functionality include:

- User discovery
  The word 'user' is ambiguous: It does not only denote a person running iServerP2P but within iServer there exists a user management part built around the abstract OM type user. The concrete occurrences of this type are either indivuduals or groups, we will preferentially use these terms to avoid confusion.
  - Instances of iServerP2P
    A precondition for communication with others is knowledge of who is available. This discovery does not have to be complete in a way that every one running the program knows every other directly, but it should provide the possiblility to reach a sensible amount of other users.
  - OM Objects Individuals and Groups
    It would be nice if the individuals or groups known to different iServer users could be synchronised/updated automatically.
- Communication
  The technical challenge how to exchange messages and data between the instances of distributed iServer program.
- Link exchange
  The main task can be subdivided into different parts including:

Marshalling: Serialization and Materialization
For transfer the links have to be serialized and at their destination it must be possible to rebuild, i.e. deserialize/materialize, the underlying OM Objects.

Query concept
Normally a user does not want to demand simply all links that other iServer instances know of but based on certain properties he wants to request specific links.

Quality: rating of received links
As a query can be fairly broad (e.g. we could simply request all links available) and even a precise query can return numerous responses (if the matching links are present in the local database of most users) the ability to identify duplicates and in general a mechanism to measure the quality of matching links should be offered.

The original task description can be found in Appendix 1.

Overview This report starts with a general review of peer-to-peer technology. Some sample applications are considered in detail. The next chapter describes the JXTA Framework. Which parts of the JXTA Java implementation are used and in which form is explained in chapter 4. Chapters 5 and 6 introduce the protocols and ideas used for user management and link exchange. Plug-in infrastructure for rating link information is introduced in the chapter 7 including several concrete examples. The central aspects concerning marshalling of user and link data are described in chapter 8. In chapter 9 the small test program used to present the functionality is shown and additionally the manual for users of iServerP2P is included. Chapter 10 gives an overview of possible alternatives and improvements. Finally, chapter 11 contains personal conclusions about this diploma project.

2 Peer-to-Peer

2.1 Definition

Peer-to-Peer (P2P) is a broad emerging topic and there does not exist a single correct definition about what makes an application or technique P2P. For practical purposes the most commonly used definitions are open ones like:

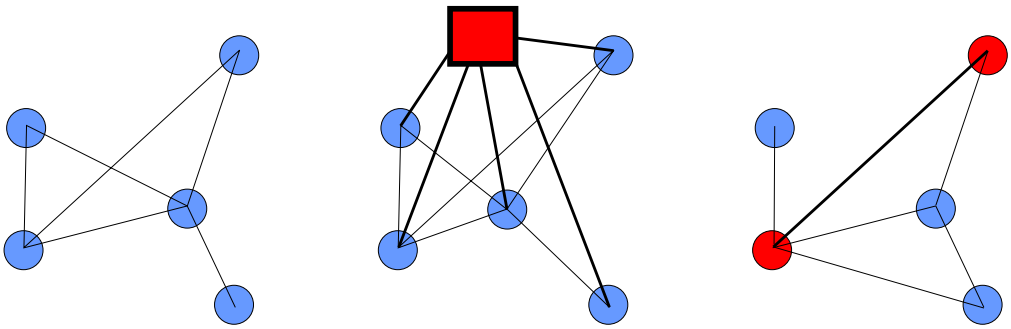
> any application or process that uses a distributed architecture and allows peers to provide and consume resources [OVUM, 02]

In this definition a peer is a networked device that implements one or more P2P protocols (like JXTA). A peer can be unreliable (e.g. by having temporary connectivity and addresses) and it can act autonomously to any extent.

Note that the previous definition does not imply that the architecture is completely serverless nor that there exists a direct communication path between the peers.

The conclusion that can be drawn from this definition and our goals is, that we want to extend the iServer architecture with a P2P component.

2.2 Architectures



(a) pure P2P (b) hybrid (c) super peer

Figure 1: P2P architectures

The *pure* P2P architecture shown in Fig. 1(a) is completely serverless. All peers are equal and the connectivity can be highly mutable.

A *hybrid* architecture like the one shown in Fig. 1(b) relies on one or more central instances. Even if none of the single peers is connected directly to

another peer, the application does not have to use a conventional client-server architecture but still can use P2P principles (see section 2.4).

An architecture using *super peers* as presented in Fig. 1(c) tries to unify the advantages of both, the hybrid and the pure architecture. The decision if a peer acts as super peer can possibly be taken dynamically based on reliability or available bandwidth. This super peer approach does not rely on fixed servers. Nevertheless, some of the peers play a more important role and therefore communication between non super peers can be minimised.

2.3 Bootstrap Problem

The bootstrap problem, a central point common to all P2P systems, deals with the question how a node (peer) joins the P2P network at startup time. Especially in case of a pure P2P architecture this proves to be a difficult problem since per definition there is no central server instance. Different strategies are used in current P2P implementations but they all have either specific requirements or introduce a central instance through the back-door:

- Broadcast
  If the underlying transport protocol supports a kind of broadcast (e.g. IP multicast) we can send out a broadcast query for other peers and the problem is solved if at least one other peer within the range of the broadcast is running and is connected to the P2P net.

- Fixed IP addresses
  Most P2P application use a list of IPs of computers that are known to run the software. This list can be hard-coded into the program or be mutable such that users can update it.

- Webpage to retrieve/register IPs
  A similar idea is to set up a dedicated site (webpage) where users can register their IP when they start up the program and where they can find IPs of other users for bootstraping. This register and lock-up can be done automatically.

- Local cache
  If in the past a peer has already been connected successfully to the network, we can rely on some of the peers used in a previous session still being available.

The bootstrap problem is discussed in [BOOTSTRAP, url] (Wiki page) and the conclusion is that no pure P2P system can be built reasonable on top of today's internet.

2.4 Application Range

There is a huge amount of P2P software available in different fields. Some examples are:

Distributed Computing	
Realtime Communications (IM)	
File Sharing	  
Groupware	

Table 1: P2P applications

Distributed Computing

Several interconnected computers share computing tasks assigned to the system. The most popular example in this area is SETI@home. As this program works fully centralised — a server hands out computing tasks to connected peers — it represent a restricted version of P2P where there does not exist direct communication between the peers. Nevertheless, the server distributes jobs based on the work other peers have done resulting in a very basic distributed architecture.

Realtime Communication and Instant Messaging

Enables online users — people or applications — to communicate "immediately" i.e. with minimal delay. A representative of realtime communication is Jabber, where each client is connected to at least one server. The most interesting distributed componont lies within the relationship between Jabber

servers: Every Jabber server is a peer to every other Jabber server. Because of its two-tiered hierarchy this can be one of the few P2P applications where the nodes (Jabber servers) operate inside the Domain Name System (DNS) and use DNS records to locate each other — but the communication between the server and the client does not rely on DNS. (see section 2.5.1 for more information about DNS)

File sharing

Filesharing is the prevalent usage of P2P and there exist many systems with different advantages and properties.

Napster for example uses the hybrid architecture shown in Fig. 1(b). All searches for files run over a central server but the payload, the file content, is transmitted over a direct connection between the peers. The advantage of this approach is that it separates the search process which needs relatively low bandwidth but is difficult to implement in a distributed way from content exchange that can easily use a direct connection.

BitTorrent is optimised for speed. It does not allow to search for content but for each file there is a central instance (tracker) that has to be known and that manages up and downloads for all users connected at a time. This priciple limits the amount of available files to those hosted actively at a time but it involves only a small protocol overhead and fairness can be easily implemented as the tracker has full control over who transmits where at what bandwidth.

Freenet enables anonymous content distribution. It is not mainly a file sharing system but arbitrary content (i.e. HTML pages) can be stored and be retrieved by anybody who knows its indentifier.

Groupware

Software that tries to integrate all of the above functionalities within a consistent and productive environment.

2.5 Example Applications

This section presents enlightening examples that show specific problems and properties of P2P applications.

2.5.1 P2P Telephony: Speakfreely

P2P Telephony applications take the word peer-to-peer literally: Every peer can contact every other. The distributed architecture that makes this possible can be very simple as in the example of speakfreely [SPEAKFREE, url].

This application uses a single central 'Look Who's Listening' facility, a directory entry on a server that matches an e-mail address, a permanent resource to identify a user, to an IP address. Once a speakfreely user has registered it IP and e-mail addresses it can be found and contacted by any other user. Each time a user changes IP address the entry on the server has to be updated.

Speakfreely has its roots already in 1991. Today it supports the Internet Real-Time Transport Protocol that runs on top of UDP/IP. Further features of speakfreely are GSM compression that allows real-time audio with a bandwith of only 1650 bytes/s and support for strong encryption. Several security issues have been identified, multiple buffer overflows permit attackers to gain system access in version 7.x.

The example of Speakfreely shows a simple method how DNS can be bypassed. DNS based approaches are impractical in todays P2P systems as changes in the translation between domain names and IP addresses take a long time to propagate. In the advent of the internet, when a permanent IP address was standard for internet users, this did not affect reachability and in the future, if IPv6 becomes widely accepted, this problem should disappear. But today it is impossible to associate one peer fix with one IP address. Therefore, in general, P2P systems can not rely on DNS because accessing decentralized resources means operating in an environment of unstable connectivity and unpredictable IP addresses [SHIRKY, 00].


On January 15th, 2004 Speak Freely development and support was discontinued by its original author, John Walker. On the webpage an *End of Life Announcement* can be found and in the paper *The Digital Imprimatur. How big brother and big media can put the Internet genie back in the bottle.* [WALKER, 02] various P2P related problems are addressed. A pessimistic developement is predicted in a strongly biased way.

P2P limitations and problems The most serious technical limitation is the increasing usage of firewalls and Network Address Translation (NAT), limiting reachability of peers shielded in that way. In addition recent broadband offers with asymmetric down- and upload bandwith limit the speed and slow down P2P applications.

Other problems that prevent the spread of P2P are factors like knowledge of users and limited availability of user friendly software. Security concerns as potential bugs in P2P software can be exploited with serious consequences. Legal doubts because copyright violations are common and can

not be prevented effectively by application developers. And finally companies and governments that not always support a community where everyone has the possibility to publish information and opinions with small effort and little or no money.

2.5.2 P2P Radio: Streamer

The basic idea of P2P radio is that every listener relays the music stream on to more listeners. In this scenario the broadcast server only needs to send the stream to few listeners in order to support an unlimited amount of listeners.

Streamer uses a webpage with a station list for finding stations. Additionally every peer submits random requests to known peers for further peers and station information. The network of all peers listening one program can be viewed as a tree where the connections are dynamically shuffled around in order to bring the higher bandwith and more reliable, i.e. having a longer uptime, relays closer to the broadcast source to act as a backbone. The low bandwith users migrate to the outer edges of the tree. A further optimisation is that feeds from inside the same Local Area Network are prefered.

P2P key advantages The example P2P Radio shows well the true power of a peer-to-peer architecture — scalability and flexibility.

2.5.3 Pure P2P search: Gnutella

Gnutella is a protocol for distributed search based on a pure P2P architecture. It is designed mainly for filesharing.

We do not consider a specific applications that implements the Gnutella protocol but the principles and ideas behind as stated in [GNUTELLA, url]. Gnutella is an example for a fully distributed architecture using simple and very basic principles. Every peer connects to relatively few others and queries are broadcasted over many indirections. This implies that such systems do not scale very well. Speed is low since a single query can potentially trigger messages for every connected peer.

For filesharing there exist improvements that scale better for a large amount of peers and frequent query broadcasts. An example is Gridella [SCHMIDT, 02], a Gnutella compatible application based on the P-Grid approach. P-Grid is a virtual search tree which is distributed among a community of peers. Randomized algorithms are used for access and search of information. All such improvements assume that the data that is available over P2P can be distributed freely over the peers. Under this restriction it

is possible to optimise the data distribution and to locate data without the need to ask every peer.

2.6 iServerP2P Requirements

In order to take the decision for one of the many P2P platforms and libraries we had to compare their features with what is needed to accomplish our goals in the most efficient way.
We need:

- User (other peers) discovery, which has to be decentral, self-controlled but not necessarily complete
- A way to send messages (queries) to all or many peers and get responses back

Nice would be:

- An existing Java implementation — as iServer is Java based — that should be opensource or at least freeware and preferably stable and tested faithfully
- The ability to exchange payload (links, user data) over the platform. That mechanism does not necessarily have to be integrated into the P2P part but could also be implemented using a direct connection (i.e. only IP distributed over P2P, data could get demanded later in the convential way)

We do not need:

- An advanced key based data sharing/storage system [1]
  Such techniques are based on the concept of distributing the data in an optimal way over all peers and then using sophisticated algorithms to locate this data without being forced to ask all peers. But we need the advantages of the simpler broadcast approach:
  1. Keep control over data distribution and redundancy: Every peer distributes only the data its user has deliberately integrated into his DB or even created himself. A query for certain data should reach a maximal (optimally all) amount of peers, and we can profit of the redundancy in the responses as important results should be contained in the DBs of many iServerP2P users and are likely to show at least once or even multiple times in the responses.

[1] sometimes called 2nd or 3rd generation P2P

2. Real search not only identification: We want to be able to properly search for arbitrary content (flexible queries) and not only to locate data based on a single property or even an unique identifier.

The next chapters show that JXTA meets all those key requirements.

3 JXTA

3.1 Technical Terms

To understand the language used to describe JXTA properties, processes and protocols a base vocabulary is required. The terms explained here are ordered by complexity starting with the most basic ones.
The thightly linked protocols are introduced in section 3.5.

- Identifier
  A peer group could potentially be as large as the entire connected universe. Naming anything uniquely is a challenge in such a large namespace. In order to cope with this, JXTA assigns an internal identifier to every addressable instance of a JXTA component. JXTA identifiers are in URN (Uniform Resource Name) format and embedded into advertisements for internal use.

  When JXTA IDs are used within protocols they are manipulated as text String URIs (Uniform Resource Identifier). In the Java reference implementation of JXTA, identification is accomplished via UUID Fields, 64-byte Strings containing numbers generated using an algorithm that ensures a high probability of uniqueness in both time and space.

- Peer
  A peer is any entity that implements the required peer protocols (JXTA core protocols). Peers are connected dynamically to other peers via pipes. There exists two kinds of super peers in JXTA:

  Relay Peer
  Maintains information on routes to other peers, and helps relay messages to peers — e.g. for firewall traversal. Note that the relay peer has ben referred to as router peer in previous JXTA versions.

  Rendezvous Peer
  A rendezvous peer maintains a cache of advertisements and forwards discovery requests to other rendezvous peers to help peers discover resources.

  Every peer in the context of a group is either a rendezvous peer or not (edge peer). In the default configuration, peers get promoted to rendezvous automatically based on criteria as connection speed, uptime or the number of other rendezvous peer in the group. (see rendezvous protocol in section 3.5)

- Peer Group
  A peer group is a collection of peers. Note that there is no concept as to why peers are grouped or how. There is also a *World Peer Group* including all visible JXTA peers. Groups can be nested. (see section 4.1)

- Peer Endpoint
  A peer endpoint is an URI (Uniform Resource Identifier) that uniquely identifies a peer's network interface (e.g. a TCP port with an associated IP address).

- Advertisement
  An advertisement is an XML document that names, describes, and publishes the existence of a peer, a peer group, a pipe, or a service.

- Message
  All information transmitted between endpoints, using pipes for example, is packaged as messages. The JXTA protocols are specified as a set of XML messages exchanged between peers. A message consists of a stack of protocol headers with bodies where the protocol body contains a variable number of bytes and one or more credentials used to identify the sender of a message.

- Pipe
  A pipe is an uni-directional, asynchronous communication channel for sending and receiving messages. Pipes are also virtual, in that a pipe's endpoint can be bound dynamically to one or more peer endpoints at runtime. The *receiving* end of a pipe is referred to as input pipe, the *sending* end as output pipe. (see Pipe Binding Protocol in section 3.5 and 4.3)

3.2 History and Principles

JXTA is a set of open, generalized XML P2P protocols that allow any connected device on the network to communicate and collaborate. It started as a research project incubated at Sun Microsystems, Inc. under the guidance of Bill Joy and Mike Clary.

> Project JXTA is short for Juxtapose, as in side by side. It is a recognition that peer to peer is juxtapose to client server or web-based computing — what is considered today's traditional computing model. [JXTAa, url]

Since February 2001 it is open source and its license is based on the Apache Software License.

Aspects such as discovery of peers, advertising presence, penetrating firewalls, and transferring data, which are common to all peer-to-peer applications, are handled by a set of standard libraries that are available to a JXTA application. JXTA allows a virtual network without a central naming or adressing authority. Every peer has a unique ID that is abstract from concrete endpoints or even transport protocols.

As JXTA uses XML as a message and advertisement format it is completely interoperable. Thanks to the simplicity and universal accessibility of XML technologies, software can be created on almost any platform to generate and parse JXTA messages. Several implementations are available, the version used for this project is the Java 2 SE reference implementation 2.1.1. (Version 2.2 was released 15.12.2003.)

3.3 Available Extensions and Applications

The development of JXTA is very active and new versions, extensions and applications appear quite frequently. We can distinguish bewteen the JXTA core, JXTA services (comparable to plug-ins in a browser) and specific JXTA applications such as iServerP2P. Some examples are listed below, many more or specific details can be found on the projects page [JXTAa, url].

JXTA services: These are extensions to the JXTA base functionality. Some of these extensions might be integrated in future releases.

- jxta-grid: Using JXTA technology for grid computing
- jngi: P2P Distributed Computing Framework
- gisp: Global Information Sharing Protocol (DHT)
- search: Distributed search service for JXTA

JXTA applications: There are not many complete applications. However, there exists a handful of commercial software based on JXTA.

- radiojxta: delivering audio content over JXTA networks
- go: A Go tournament based on the JXTA Protocols

JXTA abstractions: There exists also several approaches aiming for making programming easier. As they are all still in an early development stage and pose severe restrictions upon flexibility of usage we could not use them directly for iServerP2P but it was worth taking a look at how they implement certain tasks to have some guidelines.

- JAL: JXTA Abstraction Layer as part of EZEL (Easy Entry Library for JXTA)
- p2psockets: abstraction useful especially for developers new to P2P and being used to the conventional client-server model

3.4 Comparison of Gnutella and JXTA

Gnutella means here the underlying principles and ideas as explained in section 2.5.3. For iServerP2P we would need to put up a separate special purpose network because we are not interested in general file sharing. Table 2 gives a keyword based overview of the most important differences.

Gnutella	**JXTA**
small	huge and extensible
mainly plain file sharing	everything
unstable	attention on security
protocol only	Java reference implementation
single firewall traversal supported	communication possible if both peers are behind different firewalls
small mostly fix size messages	XML as message format (internal and payload)
payload externally	pipe abstraction for data of arbitrary size
separate networks for different application fields	groups concept

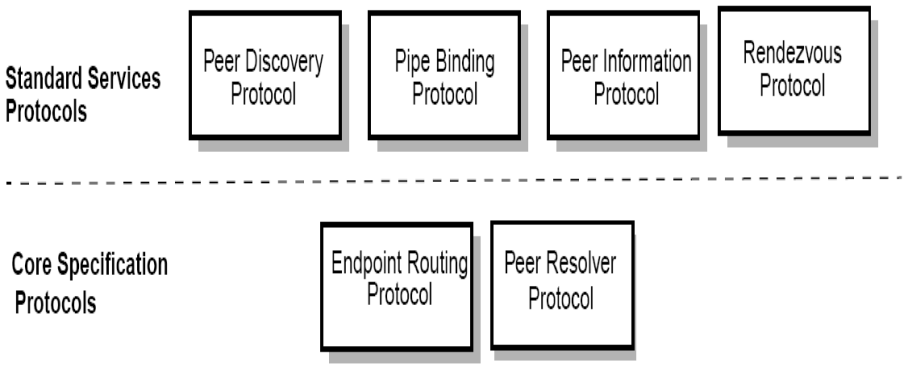
Table 2: Gnutella vs. JXTA

3.5 Protocol Stack

A considerable amount of naming and arrangement has been changed since JXTA version 1 and will probably evolve further in future versions. For example, the whole discrimination between core and standard is new. Earlier specifications considered all protocols as equal and a peer just needed to implement the protocols it required.

Project JXTA is a set of XML protocols which will be discussed in this section. Some details as how the protocols are implemented and use each other depend on the particular JXTA implementation. All our explanations are based on the Java reference implementation.

Figure 2: JXTA protocols



Core protocols

The JXTA core protocols are implemented by all JXTA peers. These two protocols and the advertisements, services and definitions they depend upon are known as the *JXTA Core Specification* and they establish the base infrastructure for P2P communication.

Endpoint Routing Protocol (ERP)/Peer Endpoint Protocol: ERP is used to ask a peer router for routes (sequence of hops) to a destination peer. Peer routers respond to queries with available route information. Any peer can decide to become a peer router by implementing the Peer Endpoint Protocol.

Peer Resolver Protocol (PRP): Peer Resolver is a generic protocol for queries to search for information or to search for peers and other JXTA items. The PRP permits the dissemination of generic queries to one or more

handlers within the group and to match them with responses. Each query is addressed to a specific handler name. A given query may be received by any number of peers in the group, possibly all, and processed according to the handler name if such a handler name is defined on that peer. A concrete example of usage is given in section 4.2.

Standard protocols

The JXTA Standard Services protocols are optional JXTA protocols and behaviours.

Rendezvous Protocol (RVP): RVP is the protocol by which peers can subscribe to or be a subscriber of a propagation service for messsages. Within a peergroup, peers can be rendezvous peers or they connect dynamically to rendezvous peers for message propagation. RVP allows messages to be sent to all of the listeners of the service. RVP is used by the Peer Resolver Protocol in order to propagate messages (see section 4.2 for consequences).

Peer Discovery protocol: Peer Discovery protocol is used to publish and discover, via advertisements, peers, peer groups, and any other advertisements.

Pipe Binding Protocol: Pipe Binding Protocol is used to bind a pipe advertisement to a pipe endpoint. Pipes are like abstract, named message queues supporting operations such as create, open, close, delete, send, and receive. The binding happens at the open operation and during close the endpoint is unbound.

Peer Information Protocol: Peer Information Protocol is used to learn about other peers' capabilities and status. This includes a ping and a name/value property access. This Protocol is not important for our application and the only one we surely can ignore.

4 JXTA in use

This chapter provides a detailed explanation of the important aspects of JXTA used in iServerP2P including some implementation details.

4.1 Groups

The iServerP2P application aims to exchange very specific information (it is not a generic filesharing or chat application). Therefore it makes sense to let this communication happen in a dedicated, newly created peer group.

We could once create a new group, named recognisable, say *iServerP2P*, for our application and from then on at startup of each new peer search on the JXTA net for a group advertisement of this name — we would find it if the advertisement still existed in the cache of a reachable peer. If we let always at least one peer connected, we could even guarantee (presumed the net is not split or completely down) that the group advertisement can be found. But if we ever wanted to update or completely change communication protocols then this approach would make it difficult to reuse the name iServerP2P. We would have to make sure not a single peer using the old version connects to our group any more. Additionally, we do not even want to publish our group so that it can be found by every JXTA user. This would just attract uninvited guests who could possible disturb our communication.

Our approach is based on a fixed group advertisement bundled with the iServerP2P application (in the `Group.adv` file). The advantage is now that each time a new version of iServerP2P is released that is not backwards compatible, we can simply generate a new group advertisement once (under the same name if we want) and then distribute this with our new iServerP2P. Of course this results in two separate communities and users running one version can no longer talk to users running a different one. In this case we could even implement bridge peers that know other versions and join several groups together.

A completely different and more elegant solution to this versioning problem is suggested in section 10.2.
Content of the group advertisement file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jxta:PGA>
<jxta:PGA xmlns:jxta="http://jxta.org">
  <GID>
    urn:jxta:uuid-D743D6515BA34CE5B300BB9B666928BF02
  </GID>
```

```
  <MSID>
    urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000010306
  </MSID>
  <Name>
    iServerP2P
  </Name>
  <Desc>
    group adv for iServerP2P application
  </Desc>
</jxta:PGA>
```

GID is the unique group identifier and MSID is the module specification ID that identifies the implementation of this group (e.g. jxta version).

4.2 Resolver

The JXTA resolver is a simple query-and-response protocol working in propagation mode. It is used internally by JXTA for discovery of advertisements.

The response to a resolver query takes the best path back to its source (rather than broadcasting). This is possible because the originating peer ID is included in the query message. The JXTA router discovers the best route back and sends the message. As the resolver is a component at a fairly low level, it is not very common to use it directly by an application. We found a single working example [BROOKb, 02] that helped a lot even if it is programmed for an older version of JXTA and several syntactical and some semantical changes were necessary in order to use it with version 2.11. The most important are:

1. `processQuery`
   This callback method for a QueryHandler executes if a resolver query arrives. Query propagation behaviour was controlled by different, more complex means in older JXTA versions.
   earlier: Throwing one of three exceptions determines if and how the query is propagated. The possible exceptions are `NoResponseException`, `ResendQueryException` and `DiscardQueryException` (for details see [WILSON, 02], page 133).
   latest: Returning an int constant is utilised now for this purpose: Either `OK success` (corresponding to no exception in the older variant) or `Repropagate` to indicate a re-propagation is needed i.e. that the query should be forwarded to the rest of the network.

2. Rendezvous Protocol
   earlier: Resolver queries are propagated to every peer.
   latest: Resolver queries are propagated only to rendezvous peers as the resolver depends on the rendezvous protocol which was changed dramatically.

The effect of the second change is, that edge peers, i.e. non rendezvous peers, will only receive direct queries for their own advertisements. The idea behind this change is, that edge peers publish indices of advertisements across the rendezvous network using Distributed Hash Tables (DHT). DHTs are maintained by rendezvous peers and queries are directed to the appropriate rendezvous ([TRAVER, 03], page 4).

As we do not have application data that can be distributed freely over the net (see section 2.6) we want to avoid this behaviour. Our workaround is to configure every peer fixed as rendezvous within the iServerP2P group. The only case where it would make sense for iServerP2P to use a peer with edge peer behaviour would be when this peer has a very slow connection or even no persistent data to be shared. In this case the peer should not receive any general queries but it should still be able to submit queries itself.

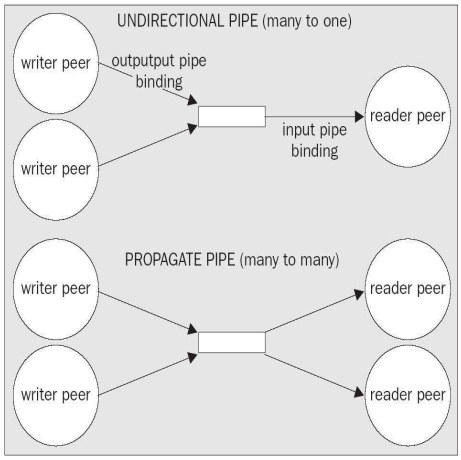
4.3 Pipe

Pipes are the core (and often the only mentioned) mechanism for JXTA peers to send messages to one each other. JXTA knows three types of pipes: point-to-point pipes, propagate pipes and secure unicast pipes. All of those default pipes are unidirectional and unreliable. In the Java implementation there exists further bidirectional and bidirectional reliable pipes built on top of the point-to-point pipe.

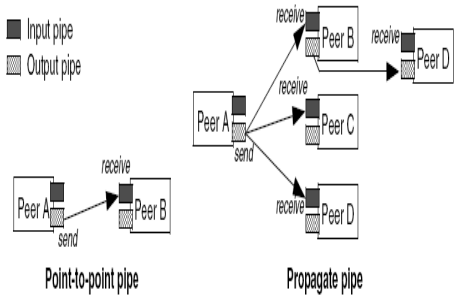
As visualised in Fig. 3 it is not obvious if several users can send messages over the same pipe, i.e. over a pipe having the same advertisement. Most literature (including the official project JXTA one) disregard this possibility but they do not state explicitely that or why this should be impossible. Discussions on the project's mailing list and our own experiments argue for the more flexible view where an arbitrary number of peers can bind an input pipe belonging to the same pipe and send messages concurrently. iServerP2P uses this feature heavily as we do not create new pipes for everyone we want to communcitate with. In our design every user listens on exactly one unidirectional point-to-point pipe and we can not predict by whom and at what times messages will arrive through this pipe.

If point-to-point pipes should ever loose their many-to-one ability, we could easily change to a policy where every peer creates a new pipe for every-

Figure 3: JXTA pipes



(a) [LI, 01], page 51

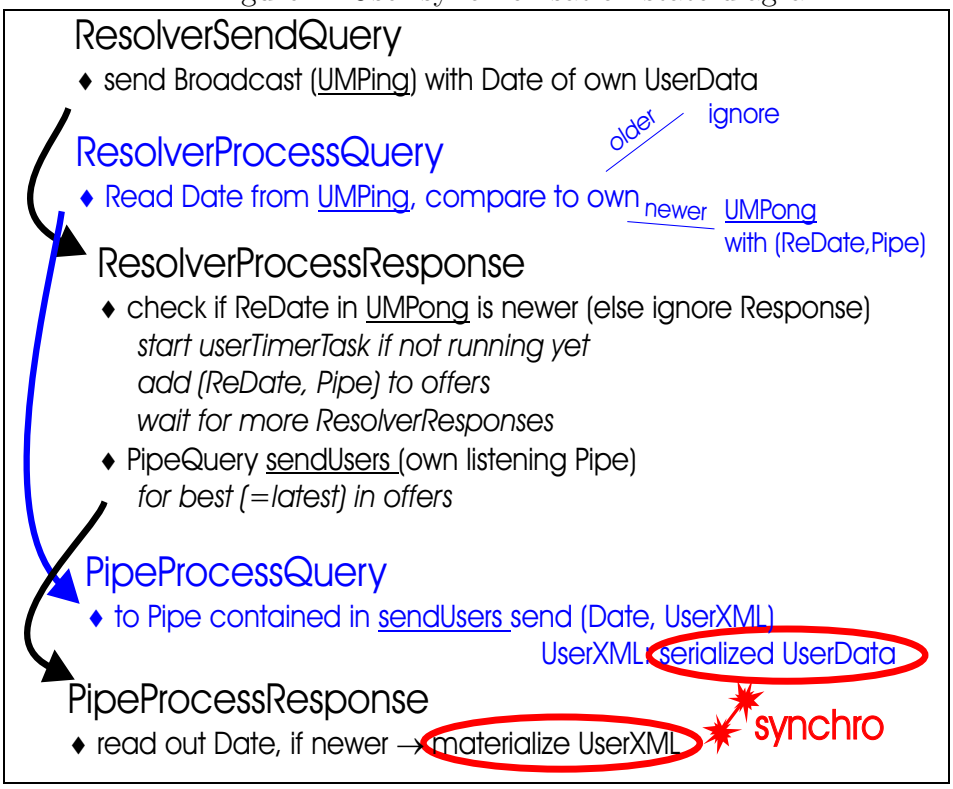


(b) [SUN, 03], page 11

one wanting to contact it as we send the corresponding advertisement anyway (over resolver communication or a pipe established earlier). In the current implementation it is just the same pipe for anybody. But this would firstly involve a large overhead and secondly it is unlikely to happen as many applications we have examined (including the chat application myJXTA, formerly known as InstantP2P, that is bundled with the JXTA Java implementation) depend on this feature.

5 User Management

Figure 4: User synchronisation state diagram



The user management[2] part of iServerP2P allows automated update of the individuals or groups (user data). The basic ideas on how this update works are the following:

- Users can never be deleted
  Individuals or groups will persist forever once created, otherwise we could not guarantee data integrity if during an update a creator of a link only we have in the database gets deleted. Further, we prevent trouble with automated update if somebody should put up a corrupt or empty version of user data.

  What we would like to have on the other hand is the ability to cancel group membership for users. In the worst case mentioned above we

[2]user management is implemented as a proof of concept rather than with the goal to be a sophisticated solution to the problem of distributed synchronisation and update

could loose membership information if we were to update to a corrupted version.

A potential problem of the impossiblity to delete users through auto-update occurs if someone should ever put up a version of user data containing a huge amount of fake data about individuals or groups. After publishing a newer fixed version, each user that already integrated those fake users would have to remove them manually.

- No incremental updates supported
  If someone wants to update his user data he has to ask for the complete list. Incremental updates affecting only users added after a certain date or only the members of a specific group are not supported.

- Only complete lists should be released
  Because of the distributed nature of the update process we can not guarantee that every iServerP2P user catches every updated version. Therefore, the latest version must always contain the complete list of relevant individuals and groups with membership information. Only administrators who can be sure to have this should put up a new version for distribution. This does not mean that other users can not create new individuals or groups in their local DB but they should not announce them directly. Local changes to group membership are prohibited as they could propagate to other users and at the time of the next auto-update they would be lost anyway.

- Versioning by date
  Each iServerP2P user knows how old the version of indivduals and groups in his DB is. Updates are only accepted if they are newer than the current version. Some plausibility checks[3] have been implemented but it is by far not invulnerable. The goal is, that every online user should be updated to the latest version sooner or later. We should definitely never "downgrade" by using an older version of user data.

Those principles are implemented in JXTA in the way shown in Fig. 4. In order to easier understand the ideas, we can abstract from the content in a first approximation and consider the user data just as an arbitrary binary large object that has a version (date) and can be updated.

The presented communication in Fig. 5 happens in a client-server (enquirer-responder) way with alternation at each step. The client is the peer that

[3]e.g. nobody can announce a version with a date years in the future, or at least it would be ignored. Otherwise anybody could render this autoupdate completely useless by distributing such a version.

wants to update its user data and sends out a query. As a server we consider a single other peer that receives this query.
The communication happens in two parts:

1. Resolver part
   The client sends out a broadcast (propagate) query over the resolver. The message is labelled UMPing and contains as a single element the version of its current user data.

   The server receives and processes this UMPing query, reads the version element it contains and compares it to the up-to-dateness of his own user data. If the server has a newer version then it responds with a UMPong message that contains the version of its user data and the advertisement of a pipe where it can be reached (by the client to asking for the content).

2. Pipe part
   If the client's request is successful and anyone managing a newer version has answered with an UMPong, it can now extract the pipe advertisement contained in the resolver answer and send a user data request to the responder (server) over this pipe. This request for user data is named sendUsers and its only content is the advertisement of the pipe where the client listens for answers.

   Any sendUsers request that arrives over a pipe is answered by sending the version of the user data and the user data itself as payload to the pipe mentioned in the sendUsers request.

   When the client finally receives such a date and userdata pair, it uses the payload data after a check if the data is still wanted.

The part not explained above is the timer component at the interspace between pipe and resolver communication. Here this is only an optimisation. As the whole user data can become arbitrary large we do not request new versions immediately but rather wait a certain time (collecting all offers that arrive in the meantime) and then choose and request the best seen so far. For link exchange as explained in the next chapter this timer is a central component that enables reasonable ranking.

The current version/date of the user data is stored externally in the file `config.xml` (see section 9.2). Alternatively this information could be stored within the iServer database, e.g. in the preferences of an individual.

Synchronization Figure 4 indicates that if we implement this design we should keep in mind that execution is not linear but asynchronous.

For the first part this does not bother us. The only relevant data is the date, which is only read in this part. So if we take care that we do never offer a version newer than the one currently used by our iServer instance we do not run into problems. This can be achieved by updating the DB always before the date.

In the second part this is more difficult. If we are about to update our user data by inserting new individuals, groups and mutate group membership our DB could temporary be in an inconsistent state. If we read the data at the same time to answer a sendUsers request we would pass on this inconsistent version.

6 Link Exchange

The ability to exchange link information with other iServerP2P users is the main goal of this project. User management as illustrated in the previous part was implemented first and in a way that requires minimal adaptions and extensions for this task. The few differences are explained in this chapter.

6.1 Queries

A first major difference in the resolver part is that the ping request now contains a specific query instead of the user data date. This query selects links the client is interested in.

The very basic language we implemented chooses a selection method based on the first three letters in the query string, the remainder is taken as argument. The following possibilities are provided:

- nam... : selects links with a given name
  Such queries make sense mainly for debug purposes.
- src... / tgt... : selects links having a specific entity as source / target
  The entity is specified by stating its XML representation. The idea is not to write such a query manually but the application should provide the possibility to select an entity to apply the query to. The XML representation can then be constructed by the serializer methods needed for communication anyway (see chapter 8).
- aql... : selects links based on the the following AQL[4] query

Because of the limitations in AQL support (and because complicated AQL query expressions can get pretty long) we could easily hard-code specific queries into the iServerP2P application. An optimal approach would probably be a dedicated *iServer Link Query Language* that would support statements containing:

- Plug-in usage for queries over data types of iServer extensions such as Paper^{++}
- Fuzzy queries as drafted in section 10.5

[4]AQL: Algebraic Query Language, the OM query language
OMS Java only supports an early version of AQL and complex expressions (nest, map, reduce, ...) are not implemented yet. Therefore, this possibility is not as powerful as we wish but queries like 'dom(HasTarget rr (Movies))' which selects links pointing to movies work.

- Structural queries referencing to interlink structures (links can have other links as source or target, a fact ignored in the current iServerP2P version) such as rings or chains of a certain length

6.2 Hash Identification

The answer to a resolver query for links should contain all matching links. Because of performance reasons (in an extreme case every connected peer finds multiple matches) we can not respond directly by sending all the data. Instead, the response is just a list of hash values, each identifying a link, together with the link's creator.[5]

A hash identifying a link is calculated using the standard Java hash function over the XML representation (as String) of the link. As the functionality to serialize a link to XML is required anyway in order to implement link exchange this approach only posed minor additional effort. One thing we had to consider here is to remove any unneccessary information (e.g. Object IDs) from the XML representation as this would lead to different hash values for the "same" links.

> Tricky issues still exist: e.g. the layer a selector lies in should be used for link exchange to have the possibility to rebuild the selector. But if this layer does not exist in the receivers database the selector should be assigned to a default layer with the result that hash identification will fail and consider the two otherwise identical links as different. A possible solution is proposed in section 10.4.

The standard Java hash function for Strings is criticised because of its relatively small hash range (32-bit integer values) and a bad distribution of values for short Strings which are mapped to small numbers. In our application small hash range should not be a problem as to a well considered query there should be a match of only few different links. Otherwise, we should probably pose a better, more specific, query. Based on the assumption of a constant, small match count the possibility of a collision is so small it can basically be ignored.

One could argue that if a typical user has a huge amount of links in his DB and if we check through hash comparison if a certain link is already known, then the hash range should be increased. Otherwise if we implemented this

[5] possible optimisation: ask for the creator of a link with a certain hash in an extra step over a pipe.
The creator would not be included in probably many responses to a broadcasted query but could be requested exactly once from a single peer knowing this link.

duplicate-check (see section 7.1) in a clever way, we would apply the query first to our own DB and then for the check we had to consider only the few matching links instead of all in our DB.

The worst case arises if two different links map to the same hash value and someone requesting a link over its hash gets a completely different one that will not even meet the query constraints. Our implementation reduces the probability of such a mix-up of links with equal hash value by storing (hashCode, link) tuples in the Hashtable `Globals.hashedLinks`. This is used as a cache that is filled every time links are offered in response to a resolver query and lookup for the link belonging to a hash is always made when someone ask for a link using its hash. The efficiency of this method depends on parameters like network traffic (how general queries arrive in what interval) or how long it takes from an arriving broadcast query until the request for a offered link is received. This problem could be eliminated if every resolver query was numbered pseudo uniquely (random) or included a sender identification. Then it would be possible to have such a cache not globally for all requesters but for each sender, and the correct cache, determined again by the ID that would also be included in the pipe request, could be used for lookup.

6.3 Resolver: Query ID

In the case of user management we only expect a single kind of answer, namely an offer for a certain version of userdata. So we do not mind if the response we receive relates to the last query we have sent or to an older one. We simply ignore the offer if it is for a version older than the one we are currently using.

For link exchange this is different. One peer can send out different queries within a short time. Therefore it is never sure to which pending query a certain offer belongs. The solution is to pass around an ID the client sends in its resolver query and which is included by the server in the resolver reply. iServerP2P leaves the management of this ID (Integer) to the application. It has to be given as parameter of the `sendQuery` method in the `LinkQueryHandler` class.

If the assigned numbers cover a large range, more memory is required because there exists a lookup table that matches the ID with the query it was last used with. Otherwise a slow response could get assigned to the wrong, more recent, query using the same ID.

6.4 Pipe Request

After some time, set through the method `setWaitingTime(int n)` of the class `LinkTimerTask`, the run method of `LinkTimerTask` is executed and there the best n links are chosen out of the pot of all offered. Chapter 7 explains in detail how links are rated. The number n can be set to a larger value than the default (1) by the method `setNumResults(int n)` of the class `LinkTimerTask`. The links are then requested one by one over pipe communication by their hash.

Only one of the probably many peers that offered a certain link is asked to send it currently — if the net proves to be unstable we should of course try a different source after waiting for an answer for a certain time.

6.5 Synchronization

Proper synchronization is even more important in link exchange than in user management. As before serialization and materialization have to be prevented from running in parallel. But here, already the construction of a resolver response relies on a consistent DB (no half-materialized links) when calculating the hash of a matching link.

7 Link Rating

All link offers are collected over a certain time. The best ones later will be requested over pipe communication.

The best offers are selected by evaluating a rating or filter function for each link. Of course, the rating function takes into account the `LinkOffer` we want to rate. Each `LinkOffer` contains the following information:

```
int hash
PipeAdvertisement adv
Individual creator
```

Additionally the rating function takes into account the complete collection of all link offers that have been received. Out of this information a broad range of different and complex rating functions can be designed.

The rating is implemented in a flexible, open way using rating plug-ins. New rating plug-ins can be added dynamically and several plug-ins together are then used to calculate a single rating value (refered to as *global rating* hereafter):

```
public void addFilter(ILinkRater r, double factor) {
    WeightedLinkRater wlr = new WeightedLinkRater(r, factor);
    filters.add(wlr);
}
```

Each rating plug-in can be given a certain weight at insertion time.

By developing classes confirming to the ILinkRater interface (see below), each application can easily write their own rating plug-ins. The only function the ILinkRater interface declares is the rating function:

```
public double rate(Vector offers, LinkOffer linkOffer);
```

To be compatible with other rating plug-ins (i.e. to allow proper weighting), this rate function should return a value between zero and one. Where one denotes a link with maximal importance and zero is assigned to a link a plug-in considers to be completely useless.

If the weight of a rater plug-in is a positive value then the result of the plug-in evaluation is added in an additive way (after multiplying with the adequate weight) to global rating.

Otherwise the result is taken into account multiplicatively, ignoring the concrete value of the weight. This allows us to use plug-ins in a blocking way — if the plug-in's rating of a link is zero the result of the global rating will also be zero independently of the value assigned to the link by other plug-ins. In this case the rater plug-in charged multiplicatively works like an additive plug-in with infinite weight.

Implementation details As can be seen in the implementation of the method `addFilter`, a Vector (filters) of separate rating elements of type `WeightedLinkRater` is used for global rating.

Figure 5: LinkRater classes

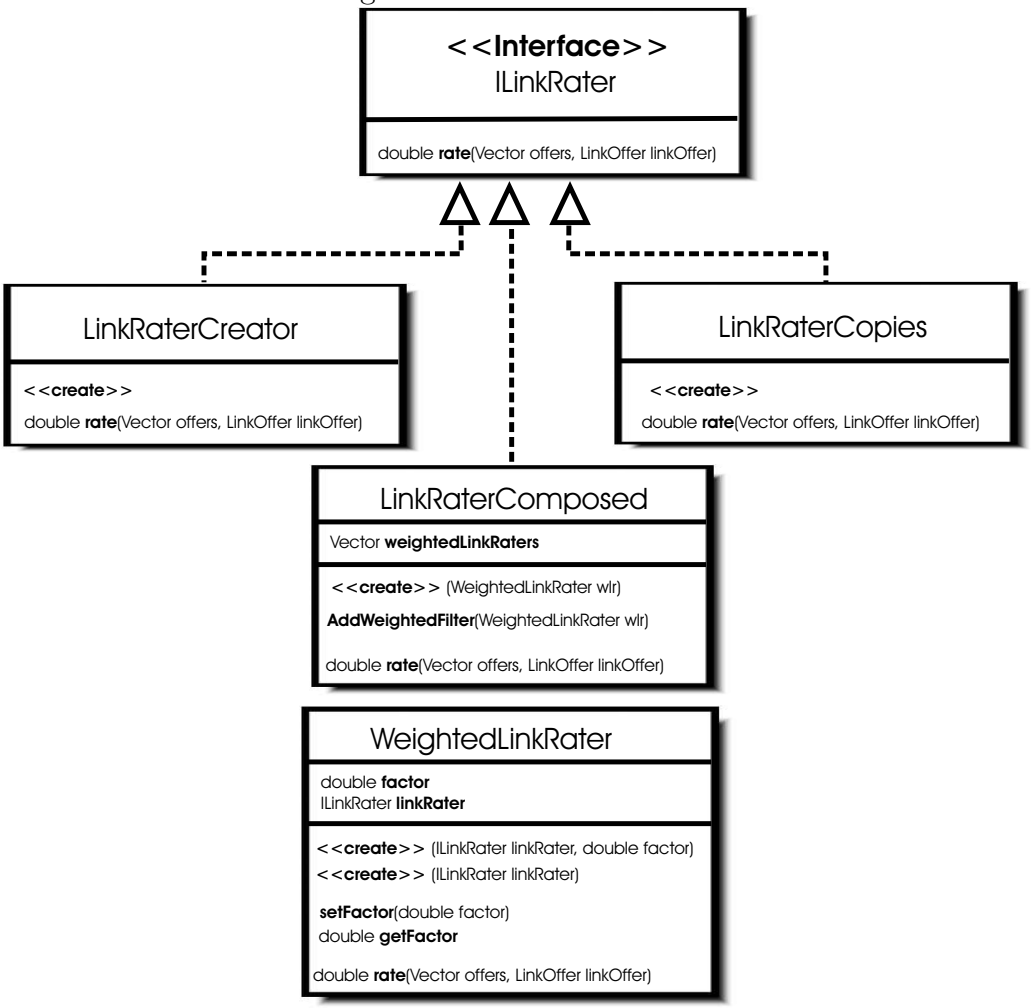


Figure 5 shows the `WeightedLinkRater` and its context. `LinkRaterBlock`, `LinkRaterCopies` and `LinkRaterComposed` are concrete examples of link rating plug-ins. The `WeightedLinkRater` binds a single `ILinkRater` to a factor that determines its weight in the context of the other rating plug-ins used. This factor can be specified at creation time, otherwise a default is used and can be changed later through the `setFactor` method.

The `WeightedLinkRater`'s rate function just relies on the rate function of the underlying `ILinkRater`.

The function that finally rates a linkOffer (global rating) is implemented as follows:

```
public synchronized double rate(LinkOffer linkOffer) {
  double result = 1;
  double factorSum = 1;
  double mult = 1;
  Enumeration enum = filters.elements();
  while (enum.hasMoreElements()) {
    WeightedLinkRater wlr = (WeightedLinkRater) enum.nextElement();
    if (wlr.getFactor() > 0) { //additive
      result = result + wlr.rate(offers, linkOffer)*wlr.getFactor();
      factorSum = factorSum + wlr.getFactor();
    } else { //multiplicative
      mult = mult * wlr.rate(offers, linkOffer);
    }
  }
  result = result / factorSum; //normalise back to range 0..1
  return result * mult;

}
```

The fact that this global evaluation scales its result back to the 0..1 range (statement between additive and multiplicative part) does not have any impact on the resulting ranking but would make sense if results of different global evaluation strategies were compared to each other.

All links with a global rationg of zero will be filtered out and never requested.

7.1 Link Rater Examples

LinkRaterCreator and LinkRaterCopies: Two pretty simple examples for rating plug-ins. LinkRaterCreator is an example of how the creator can be used for link ranking. Currently, it just rates all links created by an individual with a given name as zero all others are rated as one. Thus this plug-in can conveniently be used in a multiplicative way acting as a blacklist.

LinkRaterCopies shows how the vector of all offers can be used for ranking. It returns a value based on how often the link that is to be rated appears in the list of all offers — a clue about how popular and important this link is. Of course, if all links appear only once, all will get rated with the same

default value and this rater plug-in is ineffective. The current implementation rates more popular links higher but probably one could also imagine a scenario where a user is only interested in uncommon links and wants to rate links appearing very often lower. The decision on how flexible and adjustable such plug-ins should be is left to the application using iServerP2P.

LinkRaterComposed: The `LinkRaterComposed` class shown in Fig. 5 implements an abstract link rater plug-in composed of a bunch of `WeightedLinkRaters`. The first `WeightedLinkRater` has to be given as a parameter for the constructor and further can be added through the method `addWeightedFilter`. The `rate` function of `LinkRaterComposed` considers all underlying plug-ins (taking into account their weight). It is implemented in the same way as the global `rate` function but here the normalisation of the result to the 0..1 range is important as every ILinkRater should restrict its output range.

Further ideas for rater plug-ins: `LinkRaterDuplicates` would be a link rater that checks (by the hash) if a certain link received as answer is already known. This plug-in should be used definitely in a multiplicative way blocking known links. An optimisation could be implemented if `LinkTimerTask` was extended with a list of hashes of all matching links in the local DB. We would then compare to only those instead of all hashes. This idea could be implemented pretty easy as a new instance of `LinkTimerTask` gets created for every query anyway.

`LinkRaterConfidence` would be a very complex rating based only on the creator (see section 10.5).

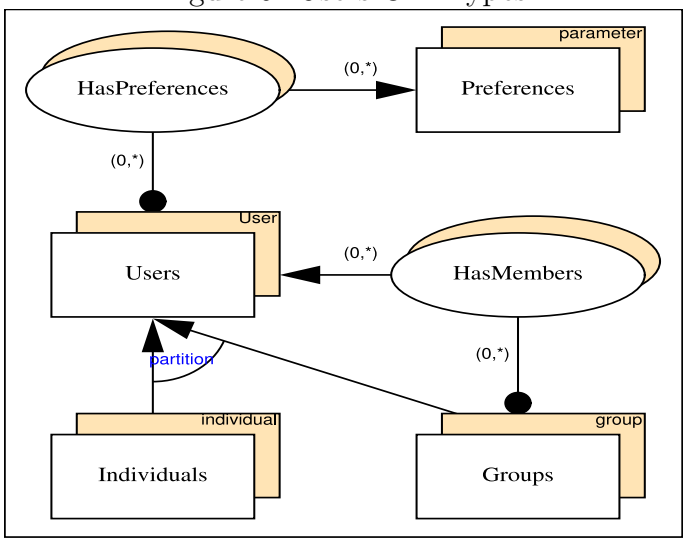
8 Serialization and Materialization

Link and user data communication both need to copy OMS Java database objects from one iServer application to the other. This data distribution should be language independent, additionally the OMS Java objects rely on the underlying database. Therefore although over JXTA pipe communication it is conveniently possible to post java objects directly this is not applicable for iServerP2P.

Instead, iServerP2P exchanges all data using an XML representation. For each data type that can be exchanged, a class exists offering the functionality to transform an object to XML and also to rebuild the OMS java objects referenced in such an XML document.

8.1 OM Types

Figure 6: Users OM Types

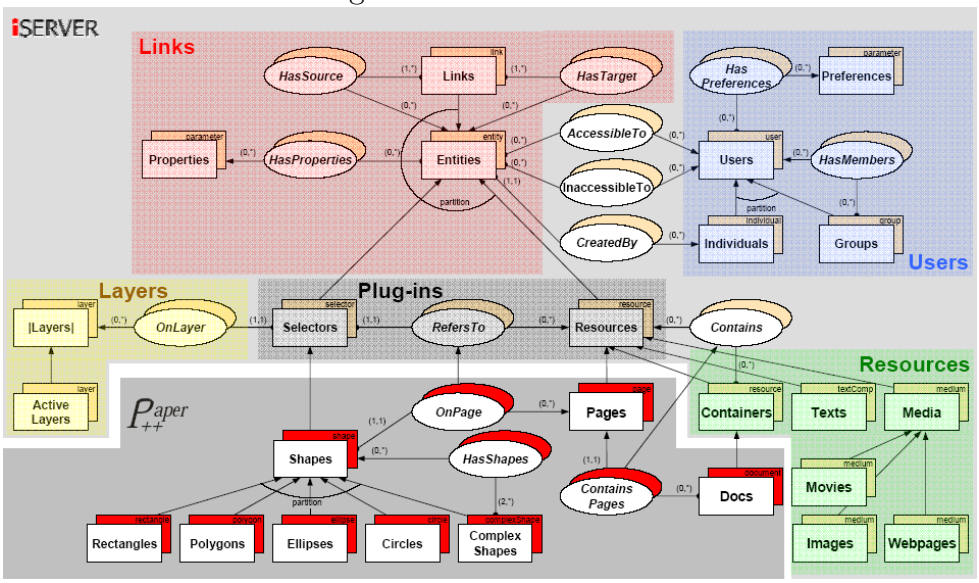


For user management only few clearly defined methods are needed as there only the two types individual and group are involved. In addition those are both subtypes of the (abstract) type user so part of the functionality needs to be implemented even only once.

For link exchange the situation is different, links can have arbitrary entities as source or target (see Fig. 7). Plug-ins are needed for every supported entity type.

This plug-in policy is very flexible as it does not specify how such XML documents are built for a certain OMS type: Mainly which parts can be

Figure 7: iServer schema



omitted because they are irrelevant to the receiver (e.g. Object ID, user preferences) or because they are not supported by iServerP2P (links as source or target). But also how certain properties should be materialized (e.g. an unknown layer of a selector should not be created newly but instead a default layer can be used). And finally design decisions about nested or flat structures, attribute or element oriented approaches and naming of attributes and tags can be made freely.

The only restriction that needs attention occurs when such marshalling relies on super type marshalling which is the case for all entities in a sensible design. Then no attributes or elements with names already used in a super type can be utilised without renaming or nesting them into extra elements. A case where this issue could easily occur is when subtypes overwrite attributes of their supertype, a feature allowed in the OM model. An example would be an entity contact that has an attribute phonenumber and a subtype private that can overwrite the phonenumber with a different one, only visible when the person is regarded the context of a private. But as every type knows its supertypes and how they are serialized this issue can be solved by renaming or nesting of elements in the XML representation.

8.2 Methods for Users

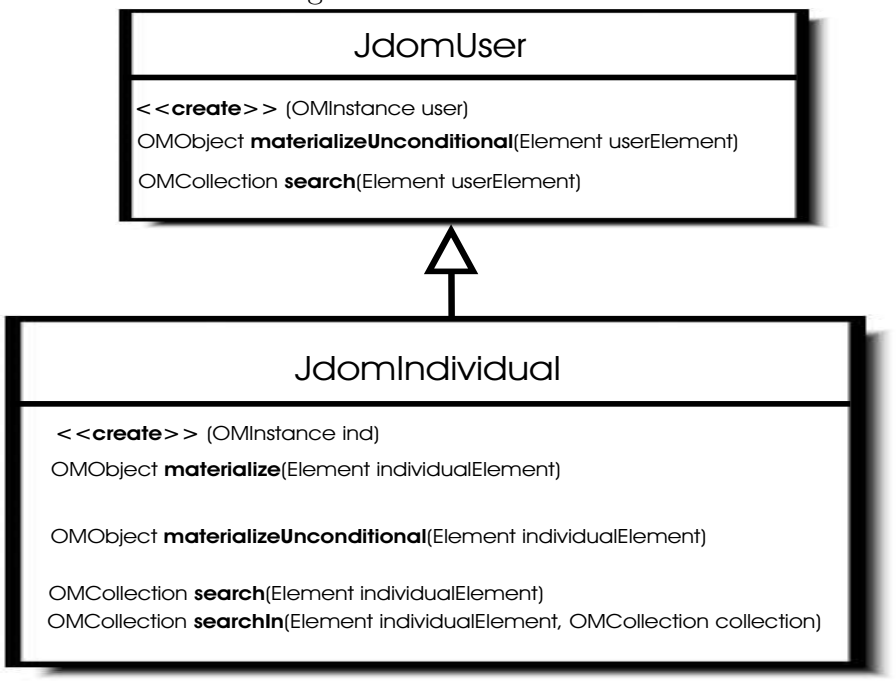
Serialization Figure 8 shows the XML representation of the simpler kind of user, an individual. The elements name and description store user attributes, login and password are specific to individuals. The oid (object ID of the OM object), the only attribute of this XML, can be left away for user management as this local property is ignored in the materialization process anyway. For link identification based on the XML the oid attribute must be omitted, as for example the creator of a link has different object IDs on different iServer instances. Also the preferences of a user are ignored by iServerP2P, they could hold application specific settings like font size or window position, information of local interest only.

Figure 8: Individual XML

```
<individual oid="OM_2032">
    <name>ch heinzer</name>
    <description/>
    <login>ch</login>
    <password>h</password>
</individual>
```

Jdom *Type* classes contain the functionality to serialize and materialize a specific *Type*.

Figure 9: JdomIndividual



All Jdom classes extend the Java type org.jdom.Element, a Java repre-

sentation of an XML Element [JDOM, url]. Figure 9 shows the two Jdom classes relevant for an individual.

The constructur of a Jdom*Type* class takes an OMInstance of type *Type* as argument and builds its Java XML representation. The reason that such a constructor does not specifiy the concrete type (e.g. individual) in its method declaration is the fact that with such an approach all Jdom*Type* constructors have the same signature and thus can be invoked generically. That is important for extensibility with future types (see section 8.5 for details).

An XML parser is used on Jdom Elements or Documents to construct a string representation that can be sent over P2P.

Materialization Materialization is more challenging because searching is involved: New users should be created only if the user does not already exists. Such a search method (static method of the Jdom*Type* class) takes a (Jdom) Element as only parameter, which is expected to contain all the necessary information.

Consider for example the steps involved in materialization of an individual received as part of an XML document (String) over P2P communication:

1. A Jdom Document is built from the String.
2. The individual Element that should be materialized is extracted from the Document.
3. `JdomIndividual.materialize` is called with the Element containing the individual information as parameter.
4. The `materialize` method first checks if such an individual is already known: `JdomIndividual.search` with the Element as parameter returns a collection of all matching individuals.
   (a) `JdomIndividual.search` relies on `JdomUser.search`
   (b) `JdomUser.search` returns a collection of users matching the user properties (name, description) of the individual.
   (c) Out of the Collection of matching users, all that are of type individual and match in the individual specific properties (login, password) are returned by `JdomIndividual.search`
5. If at least one matching individual is found, the base object of the first match is returned and materialization does not occur, otherwise the process continues with the JdomIndividual.materializeUnconditional method.

(a) `JdomIndividual.materializeUnconditional` relies on `JdomUser.materializeUnconditional`.

(b) `JdomUser.materializeUnconditional` creates the base object together with an user instance of the object, further the user attribute values (name, description) are assigned. The base object is returned.

(c) `JdomIndividual.materializeUnconditional` creates an individual instance for the base object and assignes it the individual attribute values (login, password).

6. Finally, `JdomIndividual.materializeUnconditional` returns the newly created individual's base object.

Group Specialities For groups the procedure is basically the same. The difference is that groups can have other users as members and as group is a subtype of user this structure can be deeply nested. iServerP2P treats groups different in user management and link exchange.

Groups referenced in link exchange (AccessibleTo, InaccessibleTo) are treated in a restricted way, ignoring membership. Membership information of referenced groups should not be changed for a link receiver, possibly its version of user data is more current anyway. So here only the group properties (user properties name and description and the fact that it is a group and not an individual) are taken into account for marshalling. Also group existence check (searching) only depends on those basic features and not on group members. An additional method `materializeFlat` relies on this check and if no such group is found, it creates a new one without any members.

Groups marshalled for user management contain the whole membership structure. Therefore if all users in a DB should be serialized, it will do to serialize every user not member of a group (the "roots"), otherwise redundant information would be generated.[6] Search for such explicit groups is not supported nor needed, user management uses the same existence check method as link exchange. When a group to be materialized is found to exist already, the membership information of the existing group is updated to the membership information found in the received XML.

8.3 Link Specific Issues

Link marshalling requires a mechanism to serialize and materialize arbitrary entities as the sources and targets of a link can point to anything. A restric-

[6]There is still redundancy if a group is member of several groups.

tion to this demand is implemented in iServerP2P's link serialization: all links pointed to by a certain link are ignored when serializing it. Otherwise we would run into a problem one could call "spaghetti effect": Pulling on one end can effect in raising the whole pile of thightly interlinked stuff. Applied to iServer this means if a single links should be serialized because it matches a received query this link can be connected (directly over its own sources and targets but in particularly indirectly over links connected directly) to an unforeseeable huge amount of other links, even all in the extrem case.

The implementation details about how the marshalling of arbitrary selectors and resources is handled are explained in section 8.5.

Search of a link works over its entity property (name) and the type, but as that is somewhat little information also its sources and targets (S/T) are considered but a conservative approach is implemented: S/T the sender could not serialize as it does not have a wrapper for entities of such type are not used to distinguish one link from the other. Also S/T of types the receiver does not know are ignored for existence check and for materialization.

8.4 Transient and Persistent Objects

OMS Java supports transient objects. Unlike persisten data, transient ojects exist temporary and will be disregarded at commit time, when the data is written to disk. iServerP2P makes use of this feature. All links and the referenced entities that are received over P2P are materialized as transient objects at first. The method `makePersistent` can be called on every entity and makes it and everything referenced persistent.

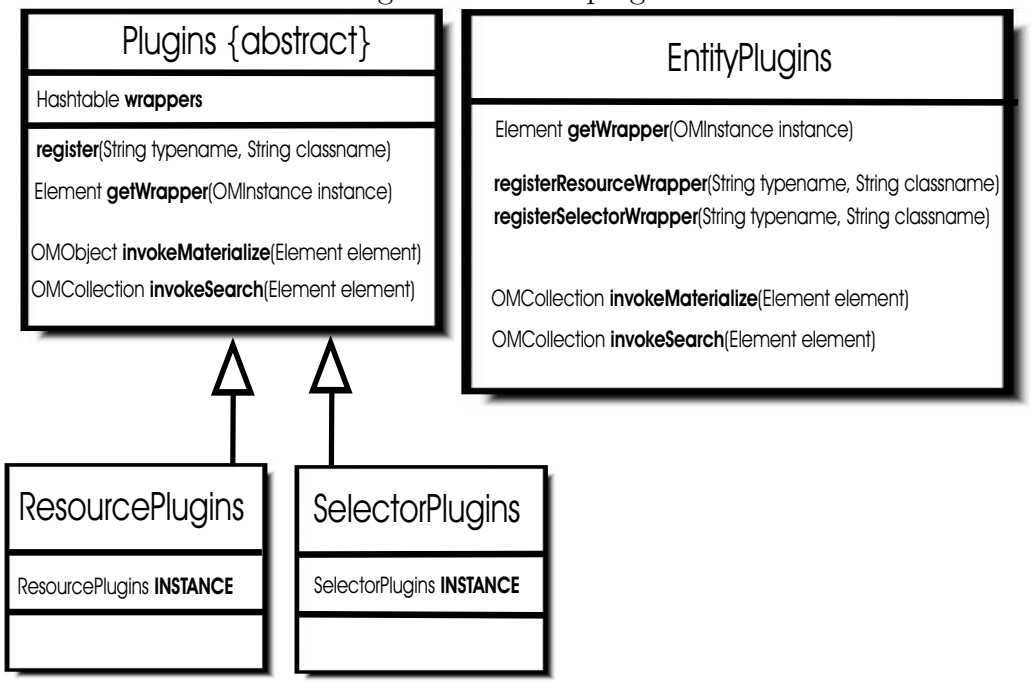
To avoid infinite loops when making mutual dependent objects persistent, the first statement in `makePersistent` for such objects should be a check if the object is already persistent.

When evaluating queries for remote peers, only persistent links are taken into account as others were received over P2P and probably never reviewed.

8.5 Plug-ins Explained

This section outlines the steps involved when a `JdomLink` is constructed and its sources and targets are serialized and how referenced entities are treated in the materialization process of a link.

Figure 10: Jdom plug-ins



Jdom plug-ins are used to locate the suitable Jdom *Type* class for marshalling of entities. The abstract class `Plugins` contains the functionality to register Jdom classes implementing this marshalling. A String indentifying a resource or selector type uniquely is associated with the name of a Jdom *Type* class responsible for this type. `ResourcePlugins` and `SelectorPlugins` are two singleton subclasses of `Plugins`, separating resouce wrappers from selector wrappers.

The constructor of `ResourcePlugins` and `SelectorPlugins` is executed at first access to the singleton object. This constructor registers the default types.

Serialization The first step for serialization of an arbitrary entity, when dealing with a links sources or targets no further type information is availabe, is implemented with help of `org.sigtec.om.util.Polymorphism.getLeaf Instance`. This method takes an OMInstance as parameter and returns the

most specific OMInstance that exists for the underlying base object. The serialization works then on this leaf OMinstance.

`EntityPlugins.getWrapper` constructs the Jdom representation for an OMInstance. This method checks if the instance is a resource or a selector and calls the `getWrapper` method of the corresponding Plugins class. The `getWrapper` method uses a `getType` method provided by each subclass of entity that returns the String identifying its type. The corresponding Jdom *Type* class can be looked up in `wrappers` and its constructor is invoked.

The XML of a resource could look like (simplified example):

```
<resource type_name="medium">
  <name>Spyglass</name>
  <content>~movies/spyglass.swf</content>
  <collection>movie</collection>
</resource>
```

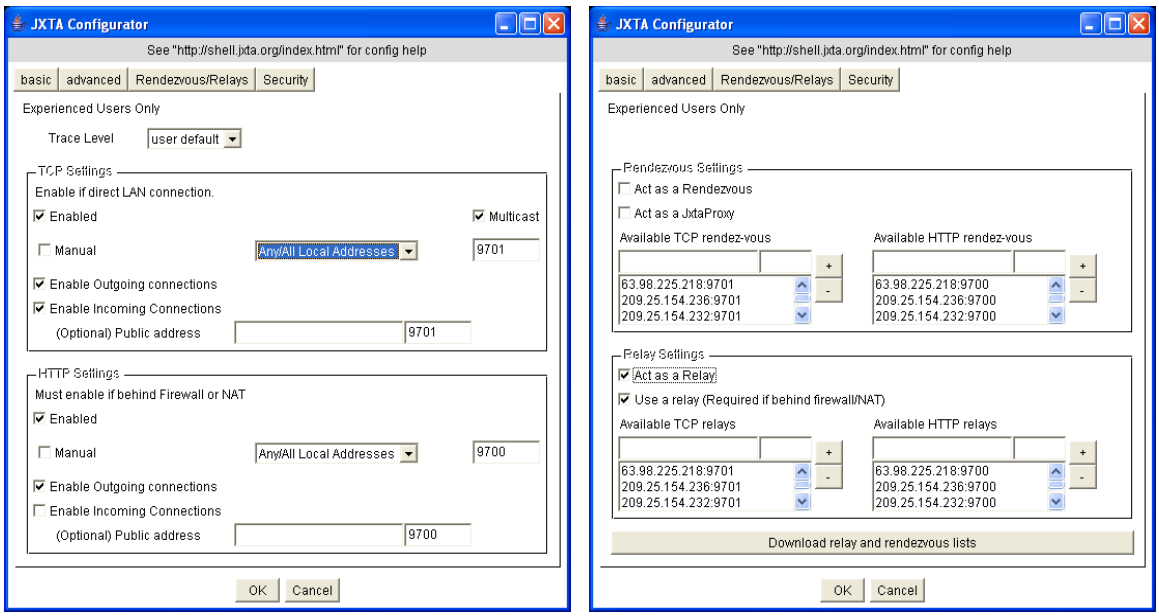
Some elements as name store entity properties and thus are present in all resource or selector XMLs, other elements and their content are resource specific. The root element is created by JdomResource where the attribute `type_name` , identifying the resource type, is set. This attribute is set using the above-mentioned `getType` method.

Materialization All information required to rebuild a resource or selector is contained in the XML. The method `EntityPlugins.invokeMaterialize` is called with a Jdom element of this XML as parameter. First is a check if this element represents a resource or a selector based on the name of the root element. The `invokeMaterialize` method of the corresponding Plugins class then looks up the correct wrapper based on the `type_name` and can invoke the static method materialize of the correct Jdom *Type* class.

9 Manual for Users and Developers

9.1 Configuration

Figure 11: JXTA configuration



(a) advanced (b) rendezvous

9.2 Startup

- OMS JAVA load the iServer DB: see OMSJava documentation
- JXTA

  ```
  import ch.ethz.inf.heinzerc.iserverp2p.ResolverPart;
  ...
  ResolverPart.INSTANCE.userTimerTask.setWaitingTime(4000);
  ```

  The first access to the singleton ResolverPart.INSTANCE triggers the startup of the JXTA platform that includes
  `config.xml`: login
  join iServerP2P Group
  `Group.adv` file, create a new if not found — communication will only be possible with peers using the *same* group advertisement. So this makes

only sense if the created file is later distributed to other iServerP2P users.

9.3 Queries and Responses

9.4 Basic Example

screenshot

9.5 Application Integration

There are two possibilities to integrate iServerP2P into existing or future applications:

1. Extend the application with startup of iServerP2P, automated user data update (send queries for new user data in a tunable interval) and finally an user-friendly method to submit sensible queries. The drawback of this approach is that source code of the application has to be available and that it takes some effort to integrate the P2P part smoothly (e.g. possibility to send a query for links on a certain entity shown on screen without the need to formulate the query explicitly in AQL).

2. Run in background a program similar to the example program shown to deposit queries. The idea is to run the P2P component as supplementary program in background and switch between base application and P2P component to send queries. The base application should then find and show any new users or links. As the current implementation of OMS Java does not support concurrent access to a single database this variant has major drawbacks. Firstly it is rather tedious as the user has to close down the base application to send queries. Secondly such utilisation will discurage users from letting always run the P2P component in background. Having peers with high uptime is very important, mainly in the introductory phase when only few peers exist, for content availability.

9.6 Troubleshooting

There are some issues and limitation we encountered and that should be known to everyone who wants to use or extend our work.

9.6.1 JXTA Startup Problems

Normally, the JXTA platform boots up without reporting any exceptions as explained in section 9.2. Nevertheless, different problems can occur:

Address already in use A `java.net.BindException: Address already in use: JVM_Bind Exception` is caused when there is another JXTA peer running on the same machine configured to use the same port.

newNetPeerGroup failed This exception was always temporarily and is presumably caused by a faulty resolver peer. If this exception should occur over an unacceptable long period of time, we advise to switch to a different resolver peer (see section 9.1).

9.6.2 java.io.IOException: Negative Seek Offset

During testing we encountered this exception several times and it is caused by a corrupt JXTA cache (on disk). The error does not completely render JXTA usage impossible but it will be thrown regularly and affects performance. The only way to get rid of this problem is to clear the cache by removing or purging the `.jxta\cm` subdirectory.
Be warned that the next startup will probably take considerably longer than normally, if the local cache is completely empty. More information about the `cm` directory can be found in [BROOKa, 02], page 89.

9.6.3 OutputPipeListener

This issue concerns the JXTA `PipeService.createOutputPipe` method which comes in two overloaded variants:

- `OutputPipe createOutputPipe (PipeAdvertisement adv, long timeout)`

  The blocking form where thread execution awaits for the creation of the output pipe (binding to the corresponding endpoint).
  The drawback of this approach is that in the worst case (if creation fails), we spend the entire timeout time until we can proceed — and possibly try to reach an alternative peer.

- `void createOutputPipe (PipeAdvertisement adv, OutputPipeListener listener)`

This variant calls back a listener when the pipe is resolved.
The advantage to the blocking form is that here several requests for outputpipe creation can be pending simultaneously.

In our original design the second approach was used but we had to realise that this does not work very reliable. When several output pipes are created back-to-back using the same `PipeAdvertisement` but always a different `OutputPipeListener` then the callback mostly fires the same listener multiple times instead of each one exactly once.

The reason for this behaviour is the way JXTA manages output pipe listeners: The class `PipeResolver` has a `Hashtable` listeners and registering a new listener via the `createOutputPipe` method in `PipeServiceImpl` results in

```
listeners.put(pipeId.toString(), listener);
```

where the `pipeId` was extracted from the `PipeAdvertisement` argument. Therefore if we try to register multiple listeners for the same `PipeAdvertisement` only the most recent is kept and will be called back later.

It is obvious now why the listener variant can only work reliable when different advertisements (from multiple peers) are concerned. Because our design was intended to work independently of the location of the communication partner and other pending communication requests, we avoided the listener variant and prefered the blocking one for busy communication where several output pipes can be created in succession.
In `LinkTimerTask.run()` and `PipeListener.SendMsg(PipeAdvertisement, Link)` the listener variant is commented out and the corresponding listener classes `LinkOutputPipeListener` and `LinkResponseSender` are marked abstract to prevent imprudent use.

10 Outlook and Alternatives

10.1 JXTA 2.2 +

10.2 Services

JXTA's core functionality is based on a service concept: Peer group services are comparable to plug-ins in a browser, they offer a certain functionality, clearly separating definition from implementation. This concept offers the advantage that e.g. third-party developers can easily define different implementations for the same functionality that can be used transparently. iServerP2P could offer its user management and / or link exchange functionality as a service and thus versioning (for updates in communication protocols) would be simplified. But even if services are rather easy to implement, adding them to a peer group and advertising them correctly is more difficult and an overhead compared to our approach where the functionality is built fix into the application.

A service always exists in the context of a specific peer group, therefore adding a new service requires creation of a new peer group and further its description as different advertisement: a module class advertisement (a rough, human readable description of the functionality), a module specification advertisement (describes the specification, all service with the same module specification are network-compatible) and finally a module implementation advertisement (describes a specific implementation of a module specification).

10.3 Pipes

only Pipe, propagate Pipe,
OutputPipe Listener instead of blocking (see 9.6.3) - implement own wrapper: based either on the asynchronous variant where only one meta listener is registered per pipe-ID and this meta listener calls back all registered listeners for the related ID. Or one could build the same functionality on top of the blocking variant just by starting a new thread for every blocking call.

10.4 Link Identification

Link identification by the help of hash functions over their XML representation was discussed elaborately and several critical aspects have been addressed. The simplest is the size of the hash used to identify links. It is only

of type int, the probability of a collision could surely be reduced if a hash of type long in association with a clever generation function was used.

All enhancements in the hash function can not solve the principal problems involved in the attempt to identify a link uniquely based on its XML representation. Some issues (default layer) could be solved by using different serializing methods for link exchange and hash generation / identification which could ignore problematic properties. The problem that occurs if different iServerP2P users knowing different types (wrappers) want to exchange links can not be solved so easily. The XML they each generate for links pointing to such types will definitely look different and this approach will identify such a link as unknown even it (with all sources and targets of known types) was just received and materialized. Perhaps this is the desired behavior because the links *do* differ in the incompatible parts but probably hashes of links already materialized once should be recorded and ignored from further responses.

A completely different approach could achieve link identification without the use of hash functions: Each link could be assigned a Globally Unique IDentifier at creation time. The distributed creation of such a GUID is not trivial, the easiest possibility is to use a huge random number as does JXTA for its identifiers. To have guaranteed uniqueness a (Time;IP) tuple could be used, link creation would then be allowed only on machines conneced to the internet over a non shared connection (otherwise we had to include the port) and only after certain time intervalls (not several new links at the "same" time). If every link was provided with such a GUID identification would be trivial. As soon as a link a user received over P2P communication was made persistent into its iServer database the decision had to be taken if this link should keep its GUID (probably if all types of sources and targets were known and could be materialized successfully), if it should keep its GUID but should not be shared (if few types of sources or targets could not be materialized and so the link lost some of its original information) or if the link should even get a different GUID (if it has been modified and can be considered as new, the receiver should be entered as creator in this case).

10.5 Link Rating

Besides hash and creator we could pack more information into the resolver response: number of sources / targets of the link, date of creation or even a date when the resources the link refers to were last checked for accessibility or correctness (this info is not available within iServer right now).
This knowledge would allow us to implement more elaborate ranking functions at the expense of a bigger size of the resolver response — something

pretty important as a single resolver query can trigger potentially one answer per reached peer.

A more challenging but possibly nicer alternative would be to leave this analysis to the supplier by packing our preferences into the query (a fuzzy query, e.g. asking for links which source includes a page near number 123 of a certain document and having as much targets as possible) and the resolver response would then contain a single matching value per link expressing how good (difficulty: all peers have to evaluate the same matching function to deliver comparable values) the demand is satisfied.

Link Ranking Based on a Confidence Value Rating Confidence values based on the creator could bring great quality improvements when a big amount of different people use iServerP2P actively authoring their own links.

We can imagine to calculate those values in a distributed way (similar to PGP) such that each user only has to judge a small amount of other users but can obtain a confidence value for many users indirectly by asking those he trusts about their opinion about a certain link creator.

This could involve matching certain peers to an individual. As it is now, those concepts are completely independent but if we want to request the opinion of a particular individual every listener of this request has to know if he is concerned.

Alternatively, every individual could have an own pipe dedicated to this confidence add-on (this pipe could be fixed for ever and stored in the Database as are login or password right now) and if someone wants to contact a certain individual to ask about his ratings, he can be reached over this pipe — of course only if at least one peer assigned to this individual is online and listening on the pipe.

The indirection mentioned above can certainly not only be a single one but can include multiple steps. Then we can not afford to update the confidence value for a particular individual every time we want to rank it. We should then keep a dynamic rating list in background and update it regularly. The idea could be:
Individuals we have rated manually keep a fixed value.
All other individuals we know hold a mutable cached value that is initialised with a neutral value at (first) startup but at some time interval we request a ranking value for a (random) individual in this cachlist from all the individuals we have rated explicitely. They answer by looking up the requested individual and probably tell us if they found it in their fix list (answer is trustable) or from their own cache (rather doubtable answer). We can then update the cachlist entry based on all the answers received and probably the

old value (makes sense if we receive only very few answers because many peers are possibly down at the moment).

If update frequency is high and most peers are not reachable for a certain time then the content of our cache will soon converge to the ranking the few reachable individuals have in their fix list. If such a feature is ever to be implemented we would propose the following procedure:

1. Analyse current usage of iServerP2P: performance, online time of different peers / users, topology of network (who knows who)
2. Predict future usage and requirements
3. Simulate this scenario and find optimal parameters for cache update frequency and mainly the implementation details of update as drafted above.

11 Personal conclusion

The goals of this project were reached and I hope that my work will be utilised and found useful.

This report points out many possible starting points for future activity and enhancements. Nevertheless the basic example that is presented works and could relatively easy be integrated into iServer applications. The entry barrier from a technical point of view is low and this report provides some valuable background information.

I can confirm the claim that JXTA is well structured and documented. Having said that this framework is huge, complex and often it was not even trivial to ignore, i.e. bypass, features I didn't intend to use at all. This fact was one of the elements that made my work exciting. The first several weeks I had no idea if I could reach my goals with JXTA or if I had to switch to a perhaps simpler but surely not so cool and feature rich alternative.

I valued the fact that the goal was well defined and concurrently I had much freedom over design and implementation decisions. That all of my OMS Java and iServer specific questions and requests could be answered by experts was yet another very positive point.

Altogether I had much fun designing, implementing and testing iServerP2P.

Acknowledgments

My thanks goes to Beat Signer who did an excellent job supervising and advising me during this project.
Special thank to Michael Grossnicklaus and Andrea Lombardoni, they supported me when problems with my profile occured or as my machine got hacked.
Further I am very grateful that Adrian Kobler — the lead programmer of OMS Java — was able to fix some required features in very short time.

References

[BOOTSTRAP, url] `http://www.cs.swarthmore.edu/cgi-bin/berney/wiki.pl?BootStrap`

[BROOKa, 02] Daniel Brookshier, Navaneeth Krishnan, Darren Govoni, Juan Carlos Soto: Java ™P2P Programming
Sams Publishing, 10.2002

[BROOKb, 02] Daniel Brookshier: JXTA Resolver
10.2002
`http://java.sun.com/features/2002/10/jxta_res.html`

[JDOM, url] `http://www.jdom.org/`

[ISERVER, url] `http://www.globis.ethz.ch/paperpp/iServer.pdf`

[JXTAa, url] Projekt JXTA *homepage* `http://www.jxta.org/`

[JXTAb, url] Projekt JXTA *protocol specification* `http://spec.jxta.org/nonav/v1.0/docbook/JXTAProtocols.html`

[GRADECKI, 02] Joseph D. Gradecki: Mastering JXTA: Building Java Peer-to-Peer Applications
Wiley Publishing, 2002

[GNUTELLA, url] `http://rfc-gnutella.sourceforge.net/developer/stable/index.html`

[LI, 01] Sing Li: Early Adopter JXTA: Peer-to-Peer Computing with Java
Wrox Press Inc, 12.2001

[NORRIE, 93] M. C. Norrie: An Extended Entity-Relationship Approach to Data Management in Object-Oriented Systems
International Conference on Conceptual Modeling / the Entity Relationship Approach, page 390-401
Springer-Verlag, 12.1993

[NORRIE, 03] M. C. Norrie, Beat Signer: Switching over to Paper: A New Web Channel
Zurich, 2003
`http://citeseer.nj.nec.com/norrie00extended.html`

[OMSJAVA, url] `http://www.omsjava.com/`

[OVUM, 02] Peer-to-Peer Computing: Applications and Infrastructure (Management Report)
Ovum Research & Consultancy, 1.2002
http://www.biz-lib.com/ZOVPP.html

[P2P, 02] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, Zhichen Xu: Peer-to-Peer Computing
HP Laboratories Palo Alto, 3.2002
http://www.hpl.hp.com/techreports/2002/HPL-2002-57.pdf

[SHIRKY, 00] Clay Shirky: What Is P2P ... And What Isn't
http://www.openp2p.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html

[SCHMIDT, 02] Roman Schmidt: Gridella: an open and efficient Gnutella-compatible Peer-to-Peer System based on the P-Grid approach
EPFL Technical Report IC/2002/71
TU Vienna, 10.02
http://citeseer.nj.nec.com/schmidt02gridella.html

[SPEAKFREE, url] http://www.fourmilab.ch/speakfree/

[STREAMER, url] http://www.streamerp2p.net/

[SUN, 03] Sun Microsystems: Project JXTA v2.0: Java Programmers Guide 5.2003
http://www.jxta.org/docs/JxtaProgGuide_v2.pdf

[TRAVER, 03] Bernard Traversat et al.: Project JXTA 2.0 Super-Peer Virtual Network
Sun Microsystems, 5.2003
http://www.jxta.org/project/www/docs/JXTA2.0protocols1.pdf

[WALKER, 02] John Walker: The Digital Imprimatur. How big brother and big media can put the Internet genie back in the bottle.
9.2003 (Revision 4, 4.11.03) http://www.fourmilab.ch/documents/digital-imprimatur/

[WILSON, 02] Brendon J. Wilson: JXTA
New Riders Publishing, 2002
http://www.brendonwilson.com/projects/jxta/

A Task Description

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institute for Information Systems:
Prof. M. C. Norrie

Diploma Thesis

Distributed iServer Architecture Based on Peer-to-Peer Concepts

Christian Heinzer

The Integration Server (iServer) architecture is an extensible cross-media linking platform enabling links between any kind of media. It is used within the European project Paper^{++}, (under the Disappearing Computer Programme, IST-2000-26130) to define links between paper and digital content and vice versa. The philosophy of the iServer architecture is to provide basic link functionality (including user management etc.) which then can be extended to support different kinds of new content (physical or digital).

The current iServer implementation stores link information within a single "centralised" database. Users can access preauthored links defined in the database as well as add their own new links which can then be shared with other users or groups of users.

The goal of this diploma thesis is to develop a decentralised version of the iServer architecture where a user stores his personal linking information in a local iServer instance. A user may access local link knowledge as well as information fetched from other users iServer databases. Since we do not know which users will be "online" at a certain time, the idea is to use similar concepts for user discovery as used in peer-to-peer networks. A user should be able to ask the community if they have linking information for a specific resource. Since the resulting set of links may be quite large we further have to classify the quality of the received links. A possible solution could be to introduce confidence values between different users which will be used for ranking.

The main tasks of this diploma thesis are as follows:

- Investigation of the current iServer framework and different existing peer-to-peer architectures.
- Definition and implementation of a user discovery service and a communication protocol to transmit linking information between different iServer instances.
- Development of a mechanism for link ranking (e.g. based on confidence values)

The project report should give a short overview over existing peer-to-peer approaches. It should then describe the protocol used to exchange information between different iServer instances and introduce one or more solutions to qualify the retrieved links. Finally, the report should provide a "user manual" and some instructions how such a distributed version of the iServer has to be installed.

Start Date:	Monday 27 October 2003
Environment:	Java, OMS Java, iServer, XML, Peer-to-Peer technology
Supervision:	Beat Signer, IFW D46.2

B API Reference

ch.ethz.inf.heinzerc.iserverp2p.ResolverPart

java.lang.Object

---

public *ResolverPart*
extends Object

ResolverPart is the entry Class for IServerP2P. As only one instance of resolverPart can exist this is a singleton Class with private Constructor. Startup of the JXTA network happens transparently in the constructor.

The methods found here could be integrated into the iServer core.

Fields

Type	Description
public static final ResolverPart	**INSTANCE**
public ResolverPart.LinkQueryHandler	**linkQueryHandler**
public ResolverPart.UserListQueryHandler	**userListQueryHandler**
public UserTimerTask	**userTimerTask**

ch.ethz.inf.heinzerc.iserverp2p.ResolverPart.UserListQueryHandler

java.lang.Object

---

public *ResolverPart.UserListQueryHandler*
extends Object

UserManagement QueryHandler for resolver.

Fields

Type	Description
protected StructuredTextDocument	**credential**
protected SimpleDateFormat	**format**

Type	Description
protected String	**handlerName**
protected PipeService	**pipe**

Methods

Returns	Description
public synchronized int	**processQuery**(ResolverQueryMsg query)
public synchronized void	**processResponse**(ResolverResponseMsg response)
public synchronized void	**sendQuery**() Query for more recent user data

ch.ethz.inf.heinzerc.iserverp2p.ResolverPart.LinkQueryHandler

java.lang.Object

---

public *ResolverPart.LinkQueryHandler*
extends Object

QueryHandler for resolver.

Fields

Type	Description
protected StructuredTextDocument	**credential**
protected SimpleDateFormat	**format**
protected String	**handlerName**
protected PipeService	**pipe**

Methods

Returns	Description
public synchronized int	**processQuery**(ResolverQueryMsg query)
public synchronized void	**processResponse**(ResolverResponseMsg response)
public synchronized void	**sendQuery**(String query, LinkTimerTask ltt, Integer id) Query for Links.

ch.ethz.inf.heinzerc.iserverp2p.UserListResponseSender

java.lang.Object

---

public *UserListResponseSender*
extends Object

Constructors

Description
UserListResponseSender() Creates a new instance of ResponseSender

Methods

Returns	Description
public void	**outputPipeEvent**(OutputPipeEvent event)

ch.ethz.inf.heinzerc.iserverp2p.EntityPlugins

java.lang.Object

---

public *EntityPlugins*
extends Object

EntityPlugins is the public Class that allows to manages Resource and Selector Plugins. A normal client should only require the two methods to register a Wrapper Everything else has to be public as it is used by Jdom classes.

Constructors

Description
EntityPlugins()

Methods

Returns	Description
public static Element	**getResourceWrapper**(OMInstance instance)
public static Element	**getSelectorWrapper**(OMInstance instance)
public static Element	**getWrapper**(OMInstance instance)
public static OMObject	**invokeMaterialize**(Element element)

Returns	Description
public static OMObject	**invokeResourceMaterialize**(Element element)
public static OMCollection	**invokeResourceSearch**(Element element)
public static OMCollection	**invokeSearch**(Element element)
public static OMObject	**invokeSelectorMaterialize**(Element element)
public static OMCollection	**invokeSelectorSearch**(Element element)
public static void	**registerResourceWrapper**(String typename, String classname) registers a resource plugin
public static void	**registerSelectorWrapper**(String typename, String classname) registers a selector plugin

ch.ethz.inf.heinzerc.iserverp2p.UserTimerTask

java.lang.Object

java.util.TimerTask

---

public *UserTimerTask*
extends TimerTask

UserTimerTask runs a certain time after the first responses to a query for user data was received. In run method, UserTimerTask chooses the best (latest) offer and gets user data over pipe communication

Fields

Type	Description
public static int	**WAITINGTIME**

Methods

Returns	Description
public void	**run**()
public synchronized void	**setWaitingTime**(int i)

ch.ethz.inf.heinzerc.iserverp2p.LinkTimerTask

java.lang.Object

java.util.TimerTask

---

public *LinkTimerTask*
extends TimerTask

A LinkTimerTask runs a certain time after a query for links was sent. In run method, LinkTimerTask chooses the best offered links and gets them over pipe communication

Fields

Type	Description
public static int	**NUMRESULTS**
public static int	**WAITINGTIME**

Constructors

Description
LinkTimerTask() Creates a new instance of LinkTimerTask

Methods

Returns	Description
public synchronized void	**addFilter**(ILinkRater r)
public synchronized void	**addFilter**(ILinkRater r, double factor)
public synchronized void	**run**()
public void	**setNumResults**(int n)
public void	**setWaitingTime**(int n)

ch.ethz.inf.heinzerc.iserverp2p.UserData

java.lang.Object

java.util.Observable

---

public *UserData*
extends Observable

Constructors

Description
UserData()
UserData(Date d)

Methods

Returns	Description
public Date	**getDate**()
public void	**processChange**()
public void	**setDate**(Date d)

ch.ethz.inf.heinzerc.iserverp2p.LinkData

java.lang.Object

java.util.Observable

---

public *LinkData*
extends Observable

Constructors

Description
LinkData()

Methods

Returns	Description
public void	**processChange**()

ch.ethz.inf.heinzerc.iserverp2p.LinkOffer

java.lang.Object

---

public *LinkOffer*
extends Object
implements Comparable

LinkOffer is a Class for a Link offered over P2P communication

Fields

Type	Description
public PipeAdvertisement	**adv** advertisement for the pipe to the peer offering this link
public Individual	**creator** creator of this link
public int	**hash** hash of the offered Link

Constructors

Description
LinkOffer(Integer hash, PipeAdvertisement adv, Individual creator) Creates a new instance of LinkOffer

Methods

Returns	Description
public int	**compareTo**(Object o) implements ordering by hash

ch.ethz.inf.heinzerc.iserverp2p.LinkRaterCreator

java.lang.Object

---

public *LinkRaterCreator*
extends Object
implements ILinkRater

Constructors

Description
LinkRaterCreator() Creates a new instance of LinkRater

Methods

Returns	Description
public double	**rate**(Vector offers, LinkOffer linkOffer)

ch.ethz.inf.heinzerc.iserverp2p.LinkRaterCopies

java.lang.Object

---

public *LinkRaterCopies*
extends Object
implements ILinkRater

Constructors

Description
LinkRaterCopies() Creates a new instance of Ex2LinkRater

Methods

Returns	Description
public LinkRater-Copies.IntegerTupel	**getMaxEquals**(Vector offers, int thisHash)
public double	**rate**(Vector offers, LinkOffer linkOffer)

ch.ethz.inf.heinzerc.iserverp2p.LinkRaterCopies.IntegerTupel

java.lang.Object

---

public *LinkRaterCopies.IntegerTupel*
extends Object

Constructors

Description
LinkRaterCopies.IntegerTupel(int maxEquals, int thisEquals)

ch.ethz.inf.heinzerc.iserverp2p.LinkRaterComposed

java.lang.Object

---

public final *LinkRaterComposed*
extends Object

implements ILinkRater

Constructors

Description
LinkRaterComposed()

Methods

Returns	Description
public synchronized void	**addWeightedFilter**(WeightedLinkRater wlr)
public synchronized void	**LinkRaterComposed**(WeightedLinkRater wlr) Creates a new instance of LinkRaterComposed
public double	**rate**(Vector offers, LinkOffer linkOffer)

ch.ethz.inf.heinzerc.iserverp2p.Globals

java.lang.Object

---

public final *Globals*
extends Object

Globals is the Class containing static members useful for several different classes within IServerP2P. This Class is final and can't be instantiated as the constructor is private. The methods found here could be integrated into the iServer core.

Fields

Type	Description
public static Hashtable	**hashedLinks** Hashtable containing (hash, Link) tuples connecting hashes offered to the corresponding link.
public static Object	**iServerDBSynchro** Object used only for synchronisation.
public static LinkData	**linkData** linkData Object can also be used for synchronisation.
public static P2PLogger	**logger** Logger to print out debug info

Type	Description
public static Object	**pipeSynchro** Object used only for synchronisation.
public static UserData	**userData** userData Object can also be used for synchronisation.

Methods

Returns	Description
public static Layer	**createDefaultLayer**() Creates a Layer called "Default". If such a layer already exists it is returned.

ch.ethz.inf.heinzerc.iserverp2p.Helpers

java.lang.Object

---

public *Helpers*
extends Object

Helpers contains assorted utility functions (all static).

Constructors

Description
Helpers()

Methods

Returns	Description
public static boolean	**dateIsValid**(Date date) Checks if a date is accepted or ignored as fake.
public static String	**dateToString**(Date date) Converts a date to it's string representation.
public static Date	**readdate**() returns the date (version) of the userdata currently in the DB
public static String	**readInputStream**(InputStream is, int len) reads a string from a InputStream.
public static String	**readTextFile**(String filename) reads a string from a file.

Returns	Description
public static String	**serializeJdomElement**(Element element, boolean plain) transforms a JdomElement to it's string representation.
public static Date	**stringToDate**(String s) Converts a string to the date it represents.
public static void	**writedate**() sets the date of userdata to the current time
public static void	**writedate**(Date d) sets the date of userdata
public static void	**writeTextFile**(String s, String filename) writes a string into a file.

ch.ethz.inf.heinzerc.iserverp2p.P2PLogger

java.lang.Object

---

public *P2PLogger*
extends Object

P2PLogger is the Class for IServerP2Ps debug output.

Methods

Returns	Description
public void	**disableOutput**() Turns logging off
public void	**enableOutput**() Turns logging on for all levels
public void	**log**(Object obj) Prints obj for debug purpose. As java logger doesn't format output nice and prints important messages twice this method uses system.out directly for now
public void	**logFinest**(Object obj) Prints obj for debug purpose, output is tagged as unimportant.
public void	**logImportant**(Object obj) Prints obj for debug purpose, output is tagged as important. Java logger will print this object twice