Modular, Service-Oriented API for Peer-to-Peer Middleware

Gábor Paller Nokia Siemens Networks
Köztelek str. 6, Budapest 1092, Hungary
gabor.paller@nsn.com

Heikki Kokkinen Nokia Research Center
Itämerenkatu 11-13, Helsinki, Finland
heikki.kokkinen@nokia.com

ABSTRACT

This paper presents a general-purpose API to P2P network middleware. The design proposes modular, service oriented solution to this problem. The paper demonstrates that while the proposed P2P API is fairly complex because it tries to support every known P2P technology, the footprint of the middleware can be quite small because of the modular, incremental nature of the API middleware. The paper also demonstrates that although service-oriented API differs significantly from the traditional, monolithic APIs, it can be implemented in small footprint on top of a non service-oriented, commercial platform, the Series60.

Categories and Subject Descriptors

D.2.11 [**Software architectures**]: Patterns—*middleware, peer-to-peer*

1. INTRODUCTION

P2P applications have become the key area of mobile application and middleware inventions because traditional, client-server middleware has matured to the point where it is very hard to come out with new solutions. P2P applications suffer from non-standard APIs and protocols. Availability of a widely accepted, sufficiently powerful API would largely help the adoption of P2P applications.

Two general trends can be recognized in the area of P2P protocols and associated middleware. One approach is *application specific*, where the protocol and the associated middleware supports solely one application type. Because the protocol itself is designed to support the needs of only the particular application type, it is very hard to run a different type of application on top of the P2P protocol and the associated middleware. P2P file sharing systems like FastTrack/Kazaa [1], Gnutella [2], eDonkey2000 [3] or BitTorrent [4] all fall into this category. APIs designed for these protocols [5] also have this restriction. The other approach is the general-purpose API that supports wide range of applications with features like messaging, pipes, service invocations, groups, real-time features, support for structured networks, etc. Offering a general-purpose API is desirable but as the widely deployed protocols are application-specific (and mostly support file sharing), general-purpose APIs need a sufficiently general P2P protocol as well. This custom protocol limits the number of the machines connected into the network - hence the value of the P2P network itself - and makes the acceptance of the general-purpose API difficult.

Thus there seems to be this paradox: application-specific P2P networks are widely accepted but can support only limited range of applications while general-purpose APIs with their support protocols do not yet form large networks therefore the value of these networks is low.

This paper describes a P2P API approach that tries to overcome this paradox. The paper proposes a modular, service-oriented API that exposes only the API features that installed P2P middleware is able to support.

The paper is organized as follows.

- Section 2 describes shortly the already existing P2P APIs.
- Section 3 presents the architecture of the API middleware and its relationship with the P2P network technology middleware.
- Section 4 presents the requirements and functionalities of our P2P API by module.
- Section 5 describes shortly one implementation of the API design on top of Symbian Series 60 platform.

2. EXISTING P2P APIS

This section presents four existing P2P APIs with different characteristics. Microsoft P2P API [6] is a general-purpose API that uses its own protocol. cP2Pc [5] is an API supporting file sharing networks. The API described in [7] provides access to structured P2P networks.

Microsoft API provides access to a very specific, Microsoft-proprietary peer-to-peer solution that is compatible only with itself. Seemingly there was no effort to incorporate other peer-to-peer protocols although the resulting API is pretty general. Main functionalities of the API are the following. *Graphing* covers the functionality of connecting

MOBILWARE 2008, February 13-15, Innsbruck, Austria

DOI 10.4108/ICST.MOBILWARE2008.2923

peers and maintaining the connections among peers. One P2P connection network is called *peer graph*. The Microsoft P2P solution allows nodes to participate in multiple graphs. Replicated graph storage is also associated to graphs. The purpose of the *Grouping API* is to allow nodes to form secure groups. The group itself and the group members are identified by secure certificates. The group creator invites group members who can then join the group and send messages to each other. The group may also have secure record store - records are signed by group members so the receiver knows that the group record item came from authorized source.

JXTA [1] is a P2P technology driven by Sun Microsystems. The technology is designed to be language- and transport protocol-neutral but in fact it does define a new protocol on top of the transport [8] to coordinate the peer-to-peer operations among the nodes. JXTA defines mappings for programming languages (e.g. Java, C). The main functionalities of the API allow the peers to discover each other, self-organize into peer groups, advertise and discover network services, communicate with each other and monitor each other. Unlike the Microsoft solution, JXTA is modularized into Platform Layer and Services Layer. The Platform Layer contains the following services: Discovery Service, Membership Service, Access Service, Pipe Service, Resolver Service and Monitoring Service. On top of the core services, JXTA offers optional services. There is one such standard service currently defined, the Shared Resource Distributed Index (SRDI).

The cP2Pc API concentrates solely on file sharing. The API supports, however, a number of P2P networks. The core API itself is quite simple: publication and revocation of single files or file collections, manipulating collections, downloading files, search for files, support functions like joining, leaving the network

Structured peer-to-peer networks create a particular structure in the overlay network while unstructured networks have an ad-hoc structure. For example a structured overlay may be organized into a virtual overlay circle [11]. Each node has a node id and the network is able to route messages toward a node ID so that the message gets closer to the target node ID with each step. Structured networks therefore are able to provide different kinds of programming abstractions than unstructured networks. The following programming abstractions are described in [7]: in Distributed Hashtable (DHT), Distributed Object Location and Routing (DOLR), group multicast/anycast (CAST).

3. ARCHITECTURE OF THE MODULAR API MIDDLEWARE LAYER

Our API middleware design work was based on extensive requirement collection phase. This resulted in an impressive list of requirements in the following areas: Network management, Monitoring, Routing, Discovery, Service invocation, Real-time data communication, Security, Identity, Power management, Resource management, Grouping, Structured networks, File sharing and Synchronization.

P2P applications range from simple file-sharing clients to complex collaboration applications accomplishing real-time communication in closed group. The protocol interoperability requirements may also vary significantly. A file sharing client is required to work with widely deployed P2P networks while a mobile application operating only on mobile clients on proximity networks is free to use a proprietary protocol. Also, widely deployed P2P networks tend to be application-oriented (like file sharing). This has the following implications.

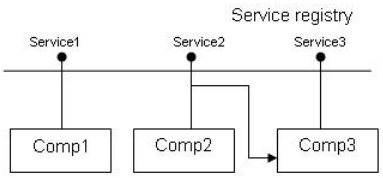

Figure 1: Service-based modular architecture

- Depending on the network technology support installed, only part of the API may be available.
- Network technology plugins may need additional middleware components to support full API functionality (e.g. group support over Gnutella network).
- Footprint of the middleware should be minimized.

A service-based, modular architecture is proposed to satisfy these requirements. We introduce the notion of *network technology plugin* which is a pluggable middleware component implementing one aspect of the expected middleware functionality. For example a middleware component implementing file sharing functionality over eDonkey2000 network can be one such network technology plugin. Network technology plugins and supporting middleware components are deployed independently and are connected by their service ports registered in a service registry. Note that at this point this is an abstract concept that can be implemented in different ways in case of different OS or runtime technologies. For example in case of Symbian, the component ports may be represented as server ports, in OSGi [2] as services, etc. This mapping is done later when the architecture is ported to the implementation platform.

The concept is depicted in Figure 1. In this example Comp1 exposes part of the API as Service1. Comp2 exposes Service2 and Comp3 builds on Service2 to provide Service3. As we are not trying to create a full-blown service model, APIs private to the P2P API are not shown. For example a Gnutella network plugin may have its own proprietary API and additional middleware is needed to adapt it to the common file sharing API. This is shown as one component exposing only the common file sharing API (Figure 2). This single component includes the Gnutella plugin and the adaptation layer needed to provide the common file-sharing API.

Network technology plugins are deployed as needed. As demonstrated later with the Symbian-based prototype, the

[1]http://www.jxta.org

[2]http://www.osgi.org

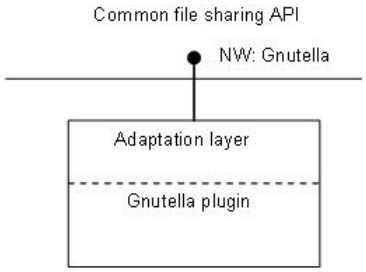

Figure 2: Private and public APIs in the network technology plugins

overhead caused by the common middleware layer is very small compared to the footprint of the original, proprietary P2P client software.

It may happen that multiple components expose the same service. For example multiple file sharing plugins may be installed. In order to be able to select the right provider (if such selection is needed) interfaces may be tagged with auxiliary meta-information. For example in the figure above the file sharing API service was tagged with the NW: Gnutella key-value pair. Service ports may be also assigned a priority. In case of multiple service port providers and no meta-info selection, the service port with the highest priority is selected. Component-level granularity may not be able to capture all the network functionality-API service combinations or would result in too many small API services that would be problematic to work with. For example Bittorrent plugin's only functionality is download that does not justify a download API with few functions. Therefore any API call may return a not implemented/not available error code which means that the particular component does not have the capability to execute that particular API function call.

Some API services may be provided by relatively few components, others, like monitoring may be exposed by many, almost all the components. In order to facilitate the programmer's job when dealing with such frequent services, the service port may be equipped with an aggregator. Aggregator is a component that provides only one instance of the specific service port and distributes the call to the underlying service ports. In order to keep our abstract service model simple, we assume the following aggregator behaviour (Figure 3).

- If the service is provided by just one component, the aggregator is not instantiated and the one service port instance is provided by the one component installed.
- If there are at least two components with the same service port installed, the aggregator is instantiated and original component service ports become internal interfaces between the aggregator and the components.

Some concerns can be encapsulated cleanly into components, other concerns crosscut other components. The API presented in this paper uses the *hook* concept to capture cross-cutting concerns. Hook is an interface designed to connect a

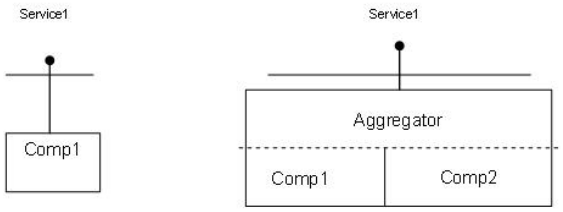

Figure 3: Service port before and after the aggregator is applied

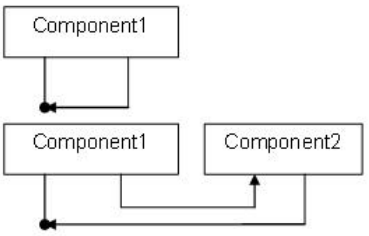

Figure 4: Component2 attached to the hook of Component1

component implementing cross-cutting concern. The hook is not supposed to be used by applications. In its initial state the hook is connected back to the component providing the hook forming "short-circuit". When a component implementing cross-cutting concern is installed, that component may intercept the hook and implement e.g. encryption features in the message flow. Figure 4 demonstrates this effect. In this figure Component1 can be seen with no hook originally then Component2 implementing a cross-cutting concern and connecting to the hook of Component1.

The hook mechanism differs from the Strategy pattern [3] in two important ways. First, hook components are not pluggable algorithms but part of the processing chain. Very often, the hook is expected to call the outgoing hook interface else the processing in the component will not continue (e.g. an outgoing packet is not generated). The other difference is that hooks are chained, if a hook does not call its outgoing interface then other hook components in the chain will not be executed.

4. API MODULES

This section describes the modules of our P2P API. [4] It is important to understand the relationship of these interfaces with middleware components that the P2P middleware is composed of. The P2P API middleware is component-based due to the large number of possible combination of middleware options installed (or not installed). Each middleware component may expose a number of interfaces of which we concentrate on public P2P API interfaces. One component may provide (and in fact, is expected to provide) several public P2P API interface modules. For example an imag-

[3]http://en.wikipedia.org/wiki/Strategy_pattern
[4]The full specification is available at http://pallergabor.uw.hu/common/p2papi_abstractspec_pub.pdf

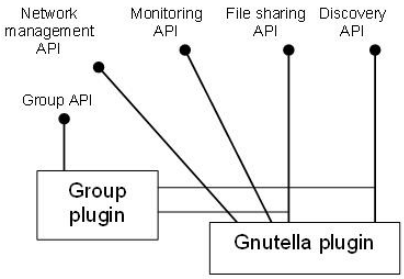

Figure 5: API modules and components

inary Gnutella plugin may provide discovery, file sharing, network management and monitoring interfaces and does not rely on any other component. Other components may rely on interfaces exposed by other components. For example an imaginary Group API component built for Gnutella networks exposes the Group API and uses the Discovery and File Sharing API exposed by the Gnutella component to publish and search for group descriptors (Figure 5).

Key property of P2P networks is that it is very hard to forecast execution times therefore most of the operations need to be asynchronous. Calling the API function just initiates the procedure; the caller is notified by an event about the outcome. The event delivery procedure may be very different depending on the actual API implementation technology, e.g. each asynchronous function may receive the callback function as parameter, listeners could be registered or there can even be an asynchronous event delivery middleware in place.

The remaining part of this section will describe the API modules shortly. Two hook interfaces may be supported by almost all the components.

Authorization hook Components call this hook when they derive an entity requesting an operation on a resource and want the authorization component to decide whether the entity has right for that operation. The hook always has the same signature; the components specify only the parameter usage.

Power management hook Power management hook is used by components to provide the central power manager with estimates how much CPU load they will cause in the foreseeable future. When the components start or stop some activity, they call this hook to update the power manager with their CPU utilization estimates. The power manager then uses these estimates to regulate CPU speed.

The main API modules and their functionalities are the following.

Network management This service is exposed by every component accomplishing network traffic. Typically these are network technology providers like Gnutella plugin but some middleware components (e.g. group middleware) may also expose this service API. The functions are: initializing, attaching to, detaching from the P2P network, setting and querying supernode status if this network technology plugin supports this kind of functionality. There is an authorization hook to handle authorization of nodes attaching to the network. These nodes open connections to some rendezvous node and the rendezvous nodes decide whether the component is allowed to attach to the P2P network.

Monitoring Monitoring service is exposed by every monitorable component, preferably every component. The service allows querying a monitoring variable identified by a key.

Discovery Discovery service is exposed by every network technology plugin that supports discovery of any resource. The resources that can be discovered are called discoverable items. Discoverable items include peers, pipe endpoints, services, files, groups, component implementations, databases and NAT [5] traversal servers. The functions are: initiating search for discoverable items, specifying conditions for discovery operation, e.g. only items in the proximity should be discovered, dispatching discovery events to discovery initiators, registering subscriptions for certain type of discoverable items, retrieving information about items discovered. There are two additional hooks supported by this interface. Authorization hook allows controlling incoming search requests. Dynamic content hook allows the component to generate item IDs from search filters received in incoming search requests on the fly.

Messaging Messaging service is exposed by every network technology plugin that is able to send a message (potentially composed of multiple network packets) from one address to another address. Address can be any node ID used by one installed network plugin in the system. Destination address is expected to be constructed according to the network technology plugin's requirements, e.g. source routing may require node ID path as destination address. This is the basic data communication mechanism providing connectionless, unreliable message sending and reception. Functions are: sending message, receiving notification about incoming message, sending network technology-supported multicast or multicast to a group. Encryption and group hooks may be supported by the network technology plugin exposing messaging service.

Pipes Pipes are built on top of the message layer. Pipes are reliable or unreliable data streams, similar to sockets in ordinary network programming. Unreliable pipes can be used to transfer real-time data as well. Real-time data streams have specific properties that may be set and respected by the network layer (e.g. that these are real-time data packets that may be discarded if there is congestion). Also network plug-ins may calculate several properties for real-time data streams important for real-time applications (e.g. jitter, delay, etc.). Pipes are identified by pipe ID and can be discovered as any other discoverable item. Functions are:

[5]Network Address Translation

binding a pipe endpoint and advertising the endpoint so that it can be discovered, opening reliable or unreliable pipe to a previously discovered pipe endpoint, doing bidirectional communication with the pipe endpoint, controlling and retrieving real-time properties of unreliable pipes. Authorization hook may be provided for incoming pipe connection requests.

Services Service API provides reliable or unreliable service invocation schemes. It is layered on top of the messaging or pipe API and serializes/deserializes unidirectional or bidirectional service invocations to and from remote services. As the actual service invocation scheme depens on the service middleware used, the P2P API takes care only of service publication, security/authentication processing and discovery. Sending and receiving service invocations is accomplished by a plugin-specific proprietary API. Authorization hook may be provided for incoming service requests.

Identity The identity provider behind the Identity API allows identification and authentication of a principal. Principal in our application domain is user or host, in some special cases it can be an application. When the identity provider is called, the aggregator behind the API calls the installed identity plugins one by one using the security parameters that were specified by the caller. During this process, one such the identity plugin checks the parameters, decides whether it is able to do such authentication (e.g. only the Liberty plugin would be able to handle Liberty authentication parameters) then performs the identification/authentication task. The result is a set of parameters stating that the principal was authenticated. These parameters can be used to invoke other APIs requiring client authentication. Identity of the current user may also be supported by the API.

Group Group service is exposed by group components or P2P middleware offering group features. The functions are the following: creating and destroying groups, joining and leaving groups, iterating over group members and installing group membership voting plugins. Group membership voting plugin is invoked when new member wants to join the group. The plugin votes on the success of join operation according to the policy (e.g. all group members have to agree, just group managers, just this node, etc.) using the credentials specified when the new member invoked the join operation. Active groups can be discovered by the Discovery API.

DHT,DOLR DHT and DOLR abstractions are supported as two separate API modules. CAST abstraction has been integrated into the group sending feature of the Messaging API. See [7] for details about these abstractions.

File sharing File sharing service is exposed by file sharing middleware. This API module is related closely to cP2Pc [5]. The functions are: publishing and unpublishing files or file collections, manipulating meta-information associated to files, search for files identified by their meta-information (this functionality is also available on the discovery service), download a file specified by its ID, suspend and resume a file download. Progress indicators related to download and other operations are available on the monitoring service API that file sharing components are encouraged to publish. Authorization hook may be provided to authorize incoming file search and download requests.

Synchronization P2P synchronization middleware is controlled by means of the Synchronization API. The API allows synchronizable data stores to be bound with each other and controlling their synchronization process. If two data stores are bound to each other, the synchronization middleware keeps them in sync if there is a connection between the nodes hosting the replicas. This means that when the nodes are connected, synchronization of the differences is performed and the changes are constantly communicated between the nodes while the nodes are connected. Synchronization may be controlled by the application or may be initiated by the middleware if specified event is received. Currently there is one such event defined, the CONNECTED event. This event is generated when connection is available between the two nodes hosting replicas of the database. Synchronizable databases can be discovered using the Discovery API.

5. P2P API PROTOTYPE ON THE SERIES60 PLATFORM

The API presented in section 4 is frighteningly large and it is indeed rather complex software with large footprint if all its features are implemented. The modular nature of the API middleware allows, however, to deploy the middleware module by module. The "core" of the API middleware has actually minimal footprint. This claim will be demonstrated on a Series60-based prototype implementation of the API middleware. Series60 is Nokia's Symbian-based smartphone platform. Series60 - as Symbian is general - is built on C++ APIs. In order to implement our P2P API design on Series60, the architecture had to be mapped to Series60 architectural element and the abstract P2P API had to be translated to Symbian C++.

The abstract architecture requires plugins that can be deployed separately. This maps well to the Series60 deployment mechanism called SIS deployer. Components have discoverable service ports tagged with meta-information. The service port abstraction is mapped to the ECom Plug-in architecture available from Series60 2.0 [9]. Multiple DLLs can expose the same interface (this is called polymorphic DLL in Symbian) and ECom allows registration, instantiation and destruction of instances behind this interface.

Multiple DLLs can expose the same interface through ECom. Selection of the DLL behind the interface is accomplished by providing a cue to the ECom framework. The cue is a text string that is matched against the properties of the DLLs. The matching is either done by the ECom framework (built-in provider) or custom provider can be implemented. ECom is suitable for our purposes because there can be multiple implementations behind one interface and interfaces can be tagged by meta-information. This maps well the service port concept. The other reason is that implementations in DLLs can be deployed separately by means of the SIS mechanism.

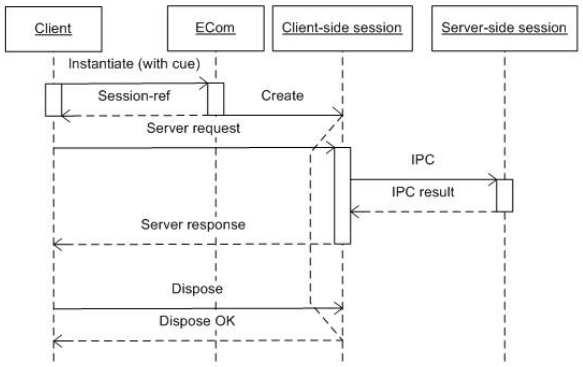

Figure 6: ECom and client-server interworking for implementing component and service abstraction

It is possible to deploy plug-ins incrementally.

There is one problem with the ECom framework: ECom objects are transient - they are created and destroyed according to client requests. P2P services are expected to execute in the background, e.g. it may be necessary to constantly respond to discovery requests or network management protocol messages, etc. For this reasons, ECom is used in conjunction with Symbian client-server framework. Symbian server is an active object - one that has its own execution state because it is executed in its own thread - that is able to accept multiple sessions from clients. The usual implementation is such that the client and server execute in separate address spaces and IPC between them is provided by the kernel by means of shared memory-based messaging. Sessions to the server are objects that have client and server-side parts. The client session object exists in the client address space while the server maintains a server-side session object in the server's address space. Figure 6 demonstrate the concept.

ECom provides simple string-based cue matching but more sophisticated cue matching can be provided by a custom resolver. The custom resolver must be able to accomplish the following tasks: assign meta-information represented by key-value pairs to components, handle interface priorities and aggregators.

The concept was implemented in a prototype. It turns out that the only overhead the P2P API middleware has is the custom resolver with some kilobytes of footprint. Other parts of the framework are integrated into the adaptation layer (figure 2) which is part of the network technology plugins. Network technology plugins may have significant footprint (as they incorporate the entire functionality of a particular P2P network access) but the adaptation layer is just a fraction of that footprint. It is still intriguing that a service-oriented, modular API could be implemented on top of a non service-oriented, commercial platform using the platform's already available mechanisms with such a small overhead.

Using a service-oriented API is slightly more complex for applications than using a traditional, monolithic API. The application must provide a search filter to obtain a reference to a suitable API module and handle the case if no provider satisifies the search filter. In our case, however, this step must be done only once because our service environment is not dynamic (we do not expect network technology plugins to appear and disappear while the application is running as in [10]).

6. CONCLUSIONS

A modular P2P API design was presented in this paper. The design tried to overcome the limitations of earlier API designs that created a false paradox between API generality and connectivity to widely deployed P2P networks. Service-oriented, modular API was proposed that has minimal footprint in small installations while is able to support complex middleware in full deployment. The approach is component-based with hook interfaces implementing cross-cutting concerns. It was demonstrated that this abstract API and its accompanying API can be implemented with minimal footprint on a non service-oriented, commercial operating system increasing the application complexity only slightly.

7. REFERENCES

[1] N. Leibowitz, M. Ripeanu, and A. Wierzbicki, Deconstructing the Kazaa Network, *3rd IEEE Workshop on Internet Applications (WIAPP'03), San Jose, CA, 2003.*

[2] Matei Ripeanu, Peer-to-Peer Architecture Case Study: Gnutella Network, *First International Conference on Peer-to-Peer Computing (P2P'01), Linköpings, Sweden, August 2001.*

[3] Oliver Heckmann and Axel Bock, The eDonkey 2000 Protocol, *KOM Technical Report 08/2002, Version 0.8, December 2002*

[4] Bram Cohen, Incentives Build Robustness in BitTorrent, *Workshop on Economics of Peer-to-Peer Systems, Berkeley, CA, USA, May 2003.*

[5] Ihor Kuz and Maarten van Steen, cP2Pc: Integrating P2P networks, *http://www.cs.vu.nl/pub/globe/cp2pc/*

[6] Introduction to Windows Peer-to-Peer Networking, *http://www.microsoft.com/ technet/ prodtechnol/ winxppro/ deploy/ p2pintro.mspx*

[7] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz and Ion Stoica, Towards a Common API for Structured Peer-to-Peer Overlays, *Peer-to-Peer Systems II: Second International Workshop, IPTPS 2003 Berkeley, CA, USA, February 21-22, 2003*

[8] JXTA v2.0 Protocols Specification, *http://spec.jxta.org/ nonav/ v1.0/ docbook/ JXTAProtocols.html*

[9] Series 60 Developer Platform 2.0: ECom Plug-In Architecture Version 1.0, *January 23, 2004*

[10] Listeners Considered Harmful: The "Whiteboard" Pattern, *Technical Whitepaper, OSGi Alliance, http:// www.osgi.org/ documents/ osgi_technology/ whiteboard.pdf*

[11] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek and Hari Balakrishnan, Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications, *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, San Diego, USA, 2001*