The Client Utility as a Peer-to-Peer System

Alan H. Karp, Vana Kalogeraki
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-2002-78
April 2nd , 2002*

E-mail: alan_karp@hp.com, vana@hpl.hp.com

Client Utility, peer-to-peer, discovery, trust, naming

The Client Utility system developed at HP Labs in the mid 1990s was designed to address the problems inherent in distributed computing. This paper shows that the architecture we developed solves some of the problems faced by designers of Peer-to-Peer systems, particularly those of discovery, trust, and naming. We show how elements of the Client Utility architecture can be used to address the problems found in some existing Peer-to-Peer systems.

* Internal Accession Date Only
Approved for External Publication

The Client Utility as a Peer-to-Peer System

Alan H. Karp, Vana Kalogeraki
Hewlett-Packard Labs
Palo Alto, California
alan_karp@hp.com, vana@hpl.hp.com

Abstract. The Client Utility system developed at HP Labs in the mid 1990s was designed to address the problems inherent in distributed computing. This paper shows that the architecture we developed solves some of the problems faced by designers of Peer-to-Peer systems, particularly those of discovery, trust, and naming. We show how elements of the Client Utility architecture can be used to address the problems found in some existing Peer-to-Peer systems.

1 Introduction

The goal of the Client Utility (CU) [1] project at HP Labs was to find a way to hide the complexity of the Internet from developers of application and users of those applications. With the increasing importance of peer-to-peer (P2P) environments [27], it has become clear that many of the issues addressed by CU are critical to the success of peer-to-peer (P2P) systems, such as Napster [12], Gnutella [14], Morpheus [13], and Freenet[15].

The most immediate problem a P2P system needs to deal with is finding out what is available or finding a specific item, be it a resource, such as a file, disk or machine, or a service of some sort, such as a backup service. A less obvious, but no less important problem, is that of trust. To what machines should you connect, and what should you allow each one to do? A third problem that is only now being recognized is that of naming things in a P2P system.

Section 2 contains a brief overview of the CU architecture, and Section 3 shows how the CU architecture addresses some of the problems common to P2P systems. Specific examples of how the approaches developed for CU can be used to address problems in P2P systems are presented in Section 4.

2 Client Utility Architecture

Although the Client Utility project started in the beginning of 1996 with its own set of goals, we ended up with an architecture that addresses the key issues we see in today's P2P environments. This section contains a brief overview of the architecture; more detail is available elsewhere [1, 2, 3, 4].

We started the Client Utility project by asking ourselves what assumptions we should make and settled on the following five.

1. Large scale. We designed for 1,000,000 machines. An environment this large means that there can be no centralized point of control and that it is impossible to maintain consistency. You must assume that any piece of data you get is out of date. If that isn't acceptable, it's up to you to synchronize.
2. Dynamic. No environment that large can be static. Machines will fail; machines will join the system. Resources, both hard, such as machines, and soft, such as files, will vanish to be replaced by new ones.
3. Heterogeneous. There is no practical way to impose a single machine architecture on the world. Not only must the system be designed to deal with different hardware and operating systems, it must accommodate devices of widely different capabilities, everything from servers to cell phones.
4. Hostile. Some people will want to wreak havoc, even if there is no gain, financial or otherwise. Given the nature of the system, the security system must be scalable. In particular, it must be able to deal with an environment consisting of an extremely large number of potential users of any resource.
5. Distributed control. The environment must be able to accommodate different ways of managing systems and expressing policies. No single standard can meet all needs nor can it be flexible enough to match the rate of change such a system will have.

It should come as no surprise that this list is identical to one we'd make today for P2P systems. To see how CU addresses each of these assumptions and how those solutions apply to P2P systems, we need to examine briefly the CU architecture.

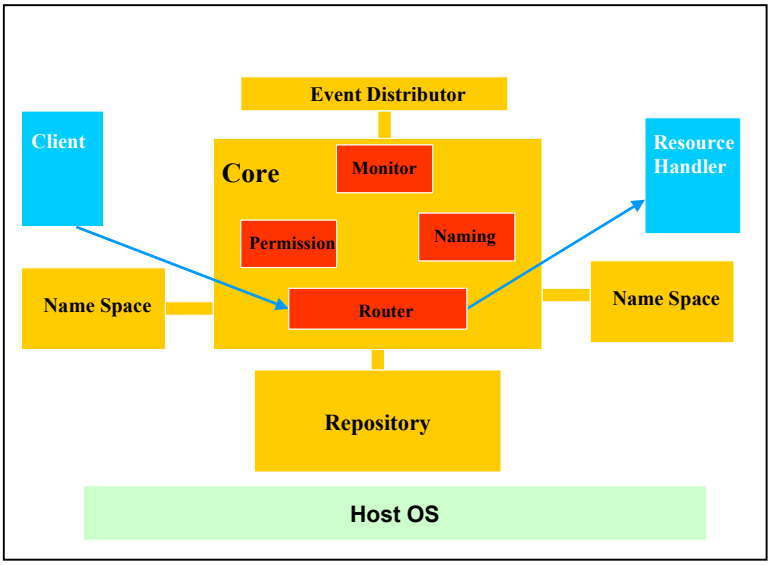

Figure 1. Message flow within a logical machine.

A CU system is a federation of *logical machines*. Each logical machine consists of an active component called the *core* and a passive component called the *repository*. Several logical machines can share one piece of hardware, each can have a dedicated machine, or a single logical machine can use several physical machines. How ever it is configured, programs will run the same way.

Figure 1 shows the key components of a logical machine. Entities that interact with the core are called *clients*. The basic unit of control is a *resource*. If a client wishes to make a resource or service available, it registers it with the core, which assigns the resource a repository handle. The corresponding repository entry designates the client acting as handler for the resource.

A mailbox metaphor is used for resource access. Each request consists of an *envelope* containing information used by the core and a *payload* containing the application related data. The message forwarded to the resource handler by the core has an envelope generated by the core and the original payload, unmodified and unexamined. This partitioning of the message means that the application API does not need modification.

The core maintains in its address space a name space on behalf of each client. This name space contains mappings between strings, representing the client's names for resources, and repository handles. Name spaces are populated in one of three ways. Some names are present when the client process first connects to the core. These names are typically bound to system resources and allow the client to bootstrap its way into the system. A second way to get a name binding is to have one sent from another client. Clients can also get name bindings by finding resources as described below.

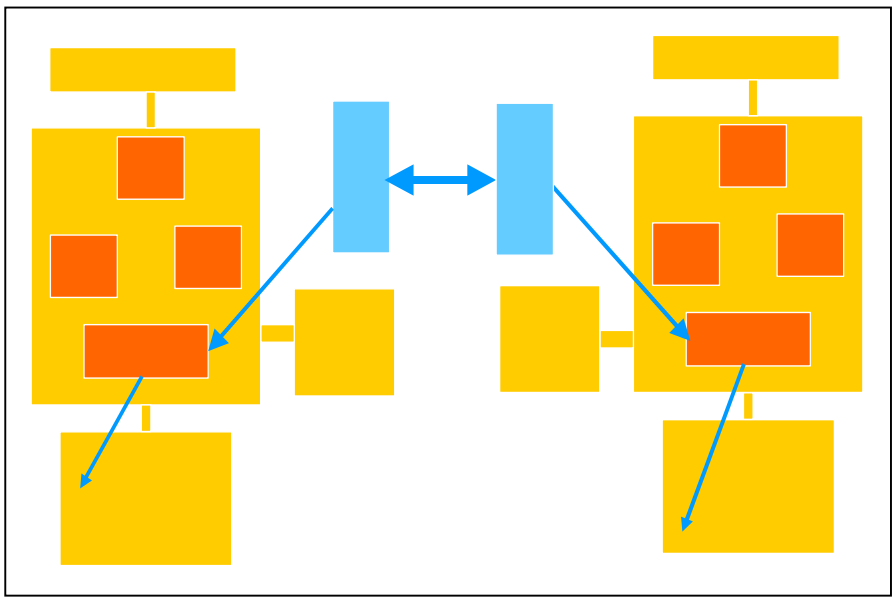

Figure 2. Connecting two logical machines.

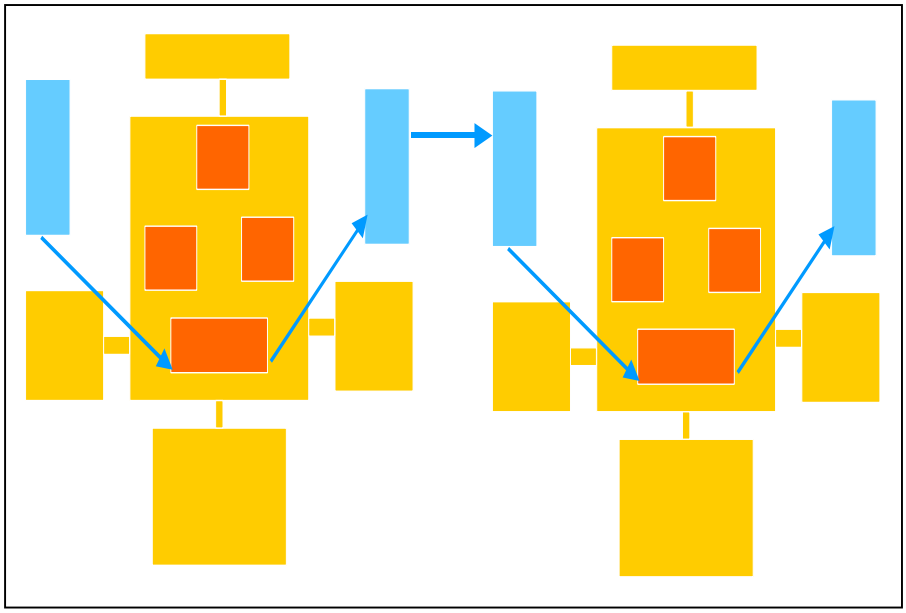

Figure 3. Message flow when accessing a remote resource.

The client message names the resource of interest. The core looks up this name in the client's name space to find the associated repository handle. The corresponding repository entry is used for permission checking and to identify the client acting as the handler for a resource. The core bundles up the original payload with a new envelope and deposits the message in the handler's inbox. Any reply is sent the same way, by naming a resource associated with the target of the message.

When a client wants to obtain access to a resource on another logical machine, it sends a request to its machine's *connection manager*. This client of the core is responsible for enforcing any policies between machines. If the connection is approved, the connection manager contacts its counterpart on the other machine, and each spawns a client to act as a proxy for the other, as shown in Figure 2. These proxies use information provided by their respective connection managers to authenticate each other and exchange encryption keys. Each then *exports* metadata of the resources it will make available to the other machine. These exported resources get registered in the repository of the importing logical machine.

When a client accesses a remote resource, the core does exactly what it does for a local resource. The only difference is that the handler is a proxy that forwards the request to its counterpart on the machine that exported the resource metadata. That proxy repeats the request exactly as issued. The request passes through the proxy's core to the resource handler. The message flow is shown in Figure 3.

Note that neither the client, nor the handler, nor either core does anything different than it does for a local request. Only the proxies are aware that there is another machine involved. Note, too that the machine on the right of the figure may not be the resource handler, only an intermediary. Or, the machine on the left may be forwarding a request that originated elsewhere. No special code is needed to do this kind of forwarding. It is a natural consequence of the basic architecture.

Figure 4 shows that the pattern of connections among CU logical machines looks very much like that found in P2P systems. Clients of a core request connections to other cores. Some of these cores act as gateways into other groups of cores. Server machines can act as intermediaries for resources from many machines. These gateway machines can also limit what connections are allowed and enforce the access control policies.

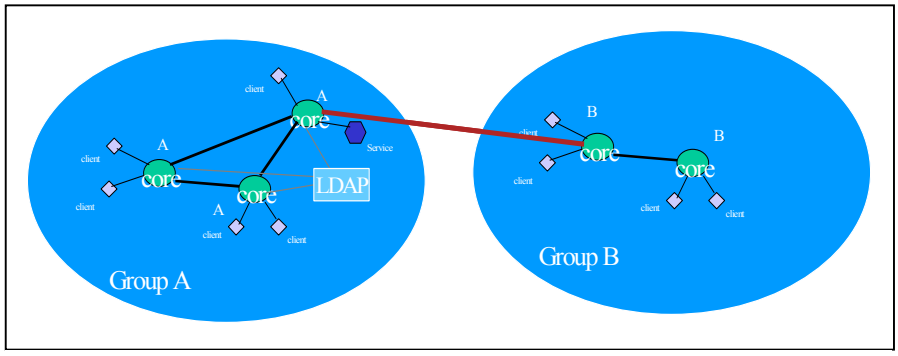

Figure 4. Scaling up the number of logical machines.

3 Issues and solutions

Designers of P2P systems need to deal with a number of issues. In this section, we'll examine questions of discovery, trust, and naming, and show how the CU architecture addresses them.

3.1 Discovery

The most obvious problem P2P systems have is finding the resource, a process we call *discovery*. The lesson learned most notably from Napster [12] is that we can't use the name of the thing to find it, even for something as simple as an immutable file. The problem is much more serious for general resources and services. The simplest approach used for services is to search for the interface of the service as done in the CORBA naming service [6] and Jini [7]. However, it is hard to see how to use this approach to find a particular instance of the many resources provided by a single service, such as a particular file in a specific file system, or to select one instance from many providers of a particular service. Another approach is to provide a common ontology to describe and discover resources as done, for example, by VerticalNet [5]. The problem is that this ontology is a standard that requires agreement to incorporate improvements and new features, but the P2P world changes too fast to wait for such agreement to be reached.

CU adopted the concept of definable ontologies to set the context for searches. It represents an ontology by a *vocabulary* consisting of attribute-value pairs. Each attribute has a number of properties, including value type, multiplicity, and a rule

defining what constitutes a match. Look-ups are performed by submitting a constraint expression of attributes from one or more vocabularies.

Vocabularies in CU are first class objects that can be described in other vocabularies. So, if you want to find a pair of shoes, you would look up shopping vocabularies in the base vocabulary defined to bootstrap the process, shoe vocabularies in the shopping vocabularies that you find, and so on. Any of the searches can return a list containing more vocabularies and other resources that match the description. So, if you've done a lookup in a men's shoe vocabulary, you may have found listings for wingtips and a vocabulary for tennis shoes. Since anyone can create a new vocabulary, someone who wanted to advertise special tennis shoes for playing on clay courts could advertise that vocabulary in the tennis shoe vocabulary; no global coordination is needed. RDF [8] can be used to provide a similar level of flexibility, but it lacks CU's definable matching rules.

The basic lookup mechanism starts with a search of the local repository. Part of the metadata stored when the resource is registered with the core is a description in one or more vocabularies. The core provides an engine for matching constraint expressions with these descriptions. If a match is found, a name binding is placed in the client's name space.

If a match is not found, the client can arrange for the search to be extended to the machines its core is connected to, much the way Gnutella [14] propagates searches. However, CU recognizes that one machine can only ask another to do something, not force it to. Hence, the CU *extended look-up request* does not specify a horizon; that's up to the machine receiving the request. Such an approach can lead to flooding the network, so requests carry a 64-bit random number as an identifying tag and an aging tag that can be used as a hint to influence a client's decision to forward the request further. This tag is only a hint, though. No matter how many hops a request has gone, a small device, such as a cell phone, may decide to forward the request if the next hop is known to be a large server. However, by taking into consideration the resources accessed by clients on different machines and exploiting their trust relationships, we can manipulate the logical connections between the machines to guarantee that peers with high degree of resource sharing are connected closely and therefore reduce the number of messages broadcast in the network [26]. If the resource is found, it is exported to the client's machine, and a name mapping to it is placed in the client's name space.

There are times when extended look-up requests aren't sufficient. For example, the client might be part of a limited community, none of the members of which has the requested resource. The extended look-up can also be inefficient, requiring too many hops and putting too much of a burden on the network and other machines. In such situations, the client can contact one of a few, well-known *advertising services*, very much like the Napster [12] model. E-speak provided one [3], but clients could use others, such as UDDI [17] or even Yahoo! [18]. Each box labeled LDAP in Figure 4 represents such an advertising service. The main difference between an extended look-

up and using the advertising service is that the former is on-line, *i.e.*, the machine owning the resource must be connected to the client by some path, but the latter is off-line. If the item is found in the advertising service, the client is told what machine to connect to.

3.2 Trust

A second issue that has not been given sufficient attention in the past but that is becoming more apparent of late is that of trust. Members of a P2P system connect their machines to each other and agree to provide certain services to peers. Each direct connection conveys a measure of trust. If this trust is misplaced, bad things happen. Note that mechanisms based on encryption, such as SSL, only prevent snooping or tampering with messages. They don't prevent one peer from attacking another over a legitimate connection.

The CU architecture was designed based on an understanding of this kind of trust. Connections are initiated and/or accepted by a connection manager, a trusted process running on each machine. The connection manager is able to associate a set of permissions with any other machine based on policies specified by the owner of the connection manager's machine. Ultimately, such policies must be set by a human, but automated tools can reduce the administrative burden. In particular, an unknown machine will only be granted a small set of privileges. CU controls these privileges so that even a rogue process, perhaps started by a virus, cannot grant excessive privileges to an attacker.

In CU, all communication with the outside world is done through a proxy running on the local machine. This policy recognizes that you trust software you install on your own machine, the proxy, more than software on machines out of your control. That doesn't mean you trust this software completely. Every CU client has a name space that limits what resources it can access and a set of permissions that limit what actions it can take on the resources it names.

The proxy serves a number of purposes. Since it is a client of the core, it has a name space. No matter what the remote user attempts to do, the proxy can only access those resources that appear in its name space and can perform only those actions allowed by its permissions. This sandboxing provides a level of control on active services similar to that provided by web servers for files. The proxy also protects against some denial of service attacks. For example, an attack that causes the proxy to crash or hang has the desirable effect of cutting off the attacker.

The transparent brokering of the CU architecture also addresses a number of trust issues. Say that Alice exports some resources to Bob, and Bob exports some of them to Carol. As far as Carol is concerned, Bob is providing the resource; as far as Alice is concerned, Bob is using the resource. Each connection represents a trust relation. Alice and Bob share one; Bob and Carol share another. There is no such relationship between Alice and Carol. Should Carol contact Alice directly and ask for the same

resource, Alice is likely to refuse. Of course, Bob could choose to introduce Carol to Alice so they can establish their own trust relation. The advantage of the introduction process is that Bob can tell Alice exactly which of Alice's services Bob is willing to forward to Carol. That information gives Alice an idea of what level of trust she should put in Carol.

3.3 Naming

Another problem, one that P2P systems share with all distributed systems, is that of naming [22,28]. Most distributed systems rely on a global name space, the most familiar of which is based on URLs. However, there are problems. One component of a URL is the location of a machine, either an IP address or a domain name. The widespread use of dynamically assigned IP addresses makes their use problematic. Domain names are better, but the presence of Network Address Translation (NAT) firewalls introduces private name spaces. Hence, it is difficult to avoid name collisions and to share names across firewalls. Changing ISPs is also a problem, since a machine's domain name must be changed. Since a global name points to a specific machine, using high availability solutions based on redundancy is hard.

CU names are purely local to each process. A translation table is maintained between each pair of entities, client-core, core-client, proxy-proxy. When a handler needs to change the name of a resource, perhaps because it was moved to a new disk, it need only inform the core, not every potential user of the resource. While this approach sounds strange, it is a system we use every day. We don't commonly use globally unique names for people and things; we use relative names. For example, instead of specifying the VIN for a vehicle, we refer to Alice's husband's car. Should Alice's husband trade in his Honda for a BMW, he may tell Alice to refer to it as his Beemer. Everyone else can still refer to Alice's husband's car. Alice can even change husbands without invalidating uses of the name.

This form of naming reflects the trust relations embodied in the connections between machines. Each hop on the path between requester and handler implies that an agreement has been reached on the meaning of the name shared by the two parties. Thus, when the handler gets a request, it knows that a "path of trust" exists; it's embodied in the name system. That assurance doesn't exist in naming systems that allow names to be communicated out of band, such as those using URLs. Those systems require an additional level of authentication before access can be granted. Path dependent names also have the advantage that the permissions are attached to the request through the names as they pass through the system. Other naming systems require a separate mechanism for access control.

Another advantage of CU's naming system has to do with the dynamic nature of large systems. There may be times when the named resource is not available, either because it has been moved, removed, or is temporarily unreachable. If the name of the object specified by the application includes location information, as it does with URLs, the resulting exception must be handled at the application level. The level of indirection

provided by CU allows the name to be remapped to an alternate provider of the same or an equivalent resource. In particular, a client's name may be bound to a list of repository handles and a lookup request. Doing so allows the core to deal with unreachable resources without involving the application.

The CU approach to naming has some limitations, of course. First of all, these names have no meaning outside of the CU system. You can't tell a friend the name of a resource over the telephone since the name has meaning only within your name space. Instead, you send a message to your friend containing the name mapping. Another problem is knowing if two names obtained from different sources refer to the same object, a problem that exists today for URLs. Most of the time it doesn't matter, say for immutable files, but when it does, CU defines a protocol that lets you find out. Clients are also free to include a large, random number in the metadata stored in the repository, allowing a high degree of certainty to the identification of any resource. Finally, there is the problem of knowing if this resource is the same one you used last time. Again, it rarely matters, but when it does, the authentication information used to establish a trust relation provides sufficient information.

4 Related Work

A great deal of work has been done in P2P systems. In this section, we'll discuss problems identified in a few of them, and show how solutions developed for CU address them.

4.1 JXTA

The JXTA project from Sun [19,20] is a "general-purpose" network programming and computing infrastructure that supports a wide range of distributed computing applications and runs on any device with a digital heartbeat. The JXTA architecture provides core functionality in multiple layers, including basic mechanisms and concepts (such as protocols for peer discovery, monitoring and group membership), higher-level services that expand these capabilities (e.g., authentication, discovery and management) and a wide range of applications (such as instant messaging and chat capabilities). There are many similarities in the goals of JXTA and Client Utility including interoperability, platform independence, discovery and security. However, JXTA focuses more on the peer groups, while in the Client Utility system the basic unit of control is the resource.

JXTA uses user-generated identifiers (UUIDs) to refer to entities and resources, such as a peer or a service. An entity has a unique UUID within the group in which it is created. This poses the important problem that the resource identities are based on human-generated names and therefore, there is no way to predict the name of a particular service. In the Client Utility, resource identities are based on attributes rather than names, which makes it suitable for searching for more general resources and services. Also, JXTA realizes the need for security and uses the resource UUIDs for implementing access control mechanisms. However, there is no guarantee that a

particular UUID won't be reused, which means that a privilege granted on the old resource may be exercised on the new one.

Client Utility, on the other hand, uses split capabilities [21], and therefore the users do not need a separate capability for each of the services or resources they invoke. Overall, the Client Utility system has developed more sophisticated and extensible mechanisms for resource identity, description and discovery based on attributes, resource access based on capabilities, and inter-machine interactions based on the use of proxies.

4.2 Oceanstore

The OceanStore system (from the University of California, Berkeley) [22] is a utility infrastructure designed to provide secure, highly available access to persistent objects in a large-scale environment. It has many attractive features including persistence storage, fault tolerance, distributed search and routing and conflict resolution. Its main design goals are: (1) to be constructed from an untrusted infrastructure and (2) to support nomadic data. There are a number of similarities in the goals of Client Utility and OceanStore - persistent storage, distribution, resource management, and secure access to resources. However, OceanStore is geared towards file sharing applications, such as groupware and repositories for scientific data. The Client Utility is not restricted to file sharing applications, but also includes legacy applications, enterprise applications, and Internet web services.

Each object in OceanStore is named by a globally unique identifier (GUID). An object GUID is the secure hash of the owner's private key and some human-readable name. The search messages are labeled with a destination GUID and routed directly to the closest node that matches the search predicate and has the desired GUID. One problem with this strategy is that objects live longer than private keys, which are often configured to expire after a year. When a private key must be changed, the names of all objects must either be changed, in which case outstanding references become invalid, or the names lose the authentication provided by using the private key in the first place.

CU on the other hand, provides a flexible and extensible naming scheme; each resource is named in the context of the local name spac,e and uses vocabularies consisting of attribute-value pairs to match the attributes of the corresponding resources. The advantage is that the client does not need to know the exact name when searching for a resource and that not all the clients have to be informed when the name (such as the local path) of the resource changes. The OceanStore system supports primitive types of access control, the reader restriction and the writer restriction. Reads are restricted at clients via key distribution, while writes are restricted at servers using access control lists. The split capabilities used in Client Utility are more flexible and make some attacks more difficult.

Ideas from the Client Utility system (such as authentication of both users and machines, privacy and access control) could be used in a system that is fundamentally untrusted. For example, CU employs a trust model with different levels of trust to grant privileges to other machines. This is superior to systems such as OceanStore that use only cryptographic techniques to protect the data.

4.3 Farsite

The Farsite project at Microsoft research [23] is a serverless, distributed file system that exploits the underutilized storage and communication resources distributed among the networked desktop computers in a large organization. Its distinguishing characteristic is that it can be deployed on an existing desktop infrastructure and does not assume careful administration or mutual trust among the client machines. Its goals are to provide high availability and reliability for file storage, security and resistance to Byzantine threats and automatic configuration and re-configuration mechanisms to respond to component failures and usage variations.

The Client Utility architecture uses a resource abstraction to encapsulate any functionality or service ranging from a file to a complex combination of services. Farsite, on the other hand, is restricted only to file systems. Each file in Farsite is encrypted (using cryptographic file hash), replicated and stored on multiple machines. The system provides security and availability by distributing multiple encrypted replicas of each file among the client machines. In terms of security, Farsite uses a directory host, which is a group of machines that interact using a Byzantine fault-tolerant protocol. This preserves the integrity of the group as long as fewer than one third of the machines misbehave in a malicious or arbitrary manner. The Client Utility started with the assumption of a malicious environment and implemented a sophisticated capability-based protection scheme. Farsite provides a good solution for the environment of a large company or university, but could take advantage of the CU mechanisms for creating, discovering and managing services across multiple organizations in a large-scale distributed system.

4.4 Collaborative P2P Platforms

Collaborative P2P platforms such as Groove 24] and Magi (Endeavors) [25] are increasingly becoming popular as they create secure shared spaces and interactive tools for real-time and asynchronous collaboration among multiple users. They use distributed state-management mechanisms that allow multiple users to share and concurrently operate application programs, while each member of the space maintains its own view. All transactions between the members of a shared space are encrypted through the use of public and private key infrastructures.

The Client Utility system, although it was not designed as a collaborative infrastructure, it is fully decentralized and has very attractive features (such as

personalizable name-spaces, scalable lookup and brokering services, access control mechanisms) that make it ideal for building large-scale P2P collaborative spaces. On the other hand, both Groove and Magi use a hybrid centralized-decentralized approach (centralized management for discovering the peers, securing access of resources and automatically updating the data, while the data is obtained directly from the peers). These are best optimized for small-group interactions.

5 Summary

Although the development of the Client Utility architecture was independent of the development of today's familiar peer-to-peer systems, CU addresses many of the problems those systems face. Some of the ideas introduced by the Client Utility have found their way into the world of Web services. These include messages consisting of envelopes and opaque payloads (SOAP [16]), mediation of requests (Biztalk [16], J2EE[16]), and extensible vocabulary structures (RDF [16]). In the peer-to-peer space, JXTA [19] has adopted many of the concepts introduced by the Client Utility. Other parts of the architecture, such as its naming system, await adoption.

The CU design, being centered on a resource abstraction, makes it an attractive platform for developing web services. In fact, its embodiment as the e-speak product [4] provided exactly such a platform. However, the basic design goals of CU are closer to those of P2P systems. Hence, as P2P developers adopt and adapt the ongoing work in web services, they may find components of the Client Utility to be useful.

5.1.1 Acknowledgements

Many thanks to Dejan Milojicic for suggesting numerous improvements to the presentation.

References

1. Alan H. Karp, Rajiv Gupta, Guillermo Rozas, Arindam Banerji, "The Client Utility Architecture: The Precursor to E-speak", HP Labs Technical Report, HPL-2001-136, (2001) available at http://www.hpl.hp.com/techreports/2001/HPL-2001-136.html
2. "E-speak Architectural Specification: Beta 2.2", http://www.e-speak.net/library/pdfs/E-speakArch.pdf (1999)
3. http://www.e-speak.net/library/pdfs/a.0/Architecture.pdf (2001)
4. Alan H. Karp, "E-speak E-xplained", HP Lab Technical Report, HPL-2000-101, http://www.hpl.hp.com/techreports/2000/HPL-2000-101.html (2000)
5. VerticalNet, http://www.verticalnet.com/
6. Object Management Group, "The Common Object Request Broker Architecture", formal/99-10-07, Version 2.3.1, October 1999.
7. Jini Network Technology, http://www.sun.com/jini
8. Resource Description Framework (RDF), http://www.w3.org/RDF
9. Simple Object Access Protocol (SOAP), http://www.w3.org/TR/SOAP

10. Biztalk, http://www.biztalk.org
11. J2EE, http://java.sun.com/j2ee
12. http://www.Napster.com
13. Morpheus 2001, The Morpheus home page, http://www.musiccity.com
14. Gnutella 2001, The Gnutella home page, http://Gnutella.wego.com
15. Freenet 2001, The Freenet home page, http://freenet.sourceforge.net
16. http://www.e-speak.net/community/esv-about.html
17. UDDI, http://uddi.org
18. Yahoo, http://yahoo.com
19. JXTA 2001, The JXTA home page, http://www.JXTA.org
20. Steve Waterhouse, David M. Doolin, Gene Kan and Yaroslav Faybishenko, "Distributed Search in P2P Networks", IEEE Internet Computing 6(1):68-72, January-February.
21. A. H. Karp, R. Gupta, G. Rozas, and A. Banerji, HP Labs Tech Report, HPL-2001-164, June (2001), http://www.hpl.hp.com/techreports/2001/HPL-2001-164.html
22. John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, R. Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao "OceanStore: An Architecture for Global-Scale Persistent Storage", Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA (November 2000).
23. John R. Douceur and Roger P. Wattenhofer, "Optimizing File Availability in a Secure Serverless Distributed File System", Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems, New Orleans, LA (October 2001), pp. 4-13.
24. Groove Networks, 2000, The Groove home page, http://www.groove.net
25. Endeavors Technology 2001, The Endeavors home page, http://www.endeavors.com
26. Murali K. Ramanathan, Vana Kalogeraki and Jim Pruyne, "Finding Good Peers in Peer-to-Peer Networks", International Parallel and Distributed Computing Symposium, Fort Lauderdale, Florida (April 2002)
27. Dejan Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins and Zhichen Xu "Peer-to-Peer Computing, HP Labs Technical Report, HPL-2002-57 available at http://www.hpl.hp.com/techreports/2002
28. Karl Aberer, Magdalena Punceva, Manfred Hauswirth and Roman Schmidt, "Improving Data Access in P2P Systems", IEEE Internet Computing 6(1):58-67, January-February.