One Ring to Rule Them All: Service Discovery and Binding in Structured Peer-to-Peer Overlay Networks

Miguel Castro
Microsoft Research – Cambridge

Peter Druschel
Rice University

Anne-Marie Kermarrec
Microsoft Research – Cambridge

Antony Rowstron
Microsoft Research – Cambridge

PDN-02-004
September 2002

Appears in *Proceedings of the SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

One Ring to Rule them All: Service Discovery and Binding in Structured Peer-to-Peer Overlay Networks

Miguel Castro
Microsoft Research,
7 J J Thomson Close,
Cambridge,
CB3 0FB, UK.
mcastro@microsoft.com

Peter Druschel
Rice University,
6100 Main Street,
MS-132, Houston,
TX 77005, USA.
druschel@cs.rice.edu

Anne-Marie Kermarrec
Microsoft Research,
7 J J Thomson Close,
Cambridge,
CB3 0FB, UK.
annemk@microsoft.com

Antony Rowstron
Microsoft Research,
7 J J Thomson Close,
Cambridge,
CB3 0FB, UK.
antr@microsoft.com

One Ring to rule them all. One Ring to find them. One Ring to bring them all. And in the darkness bind them. *J.R.R. Tolkien*

Abstract

Self-organizing, structured peer-to-peer (p2p) overlay networks like CAN, Chord, Pastry and Tapestry offer a novel platform for a variety of scalable and decentralized distributed applications. These systems provide efficient and fault-tolerant routing, object location, and load balancing within a self-organizing overlay network.

One major problem with these systems is how to bootstrap them. How do you decide which overlay to join? How do you find a contact node in the overlay to join? How do you obtain the code that you should run? Current systems require that each node that participates in a given overlay supports the same set of applications, and that these applications are pre-installed on each node.

In this position paper, we sketch the design of an infrastructure that uses a universal overlay to provide a scalable infrastructure to bootstrap multiple service overlays providing different functionality. It provides mechanisms to advertise services and to discover services, contact nodes, and service code.

1. Introduction

Recent systems such as CAN [11], Chord [15], Kademlia [8], Pastry [12] and Tapestry [17] provide a self-organizing structured peer-to-peer (p2p) overlay network that can serve as a substrate for large-scale peer-to-peer applications. One of the abstractions that these systems can provide is a scalable, fault-tolerant distributed hash table (DHT), in which any item can be located within a bounded number of routing hops, using a small per-node routing table.

In these systems a live node in the overlay to each key and provide primitives to send a message to a key. Messages are routed to the live node that is currently responsible for the destination key. Keys are chosen from a large space and each node is assigned an identifier (*nodeId*) chosen from the same space. Each node maintains a routing table with nodeIds and IP addresses of other nodes. The protocols use these routing tables to assign keys to live nodes. For instance, in Pastry, a key is assigned to the live node with nodeId numerically closest to the key.

In the simplest case, DHTs can be used to store key-value pairs much like centralized hash tables. Lookup and insert operations can be performed in a small number of routing hops. The overlay network is completely self-organizing, and each node maintains only a small routing table with size constant or logarithmic in the number of participating nodes. Structured p2p overlays can be used as a platform for a variety of distributed services, including archival stores [7, 3, 13], content distribution [6] and application-level multicast [18, 2, 14].

Service advertisement, discovery and binding are common problems in distributed systems [10, 16, 9]. Service advertisement and discovery mechanisms allow a user to deploy and find services of interest, and binding provides the user with the code necessary to install the service on a node. These problems are compounded in p2p overlays because the service is run by a large number of diverse, distributed peers. Furthermore, binding is harder for a p2p service because a joining peer is required to know a *contact node* already in the overlay.

Current p2p overlays do not provide a good solution to these problems. They require that each node supports the same set of applications, and that these applications are pre-installed on each node. Additionally, they do not provide a scalable solution to find a contact node to join an overlay.

In this position paper, we sketch the design of an infrastructure that uses a universal p2p overlay to provide scalable mechanisms to bootstrap multiple service overlays providing different functionality. It provides mechanisms to advertise and discover services, contact nodes, and service code.

In the following description, we will use Pastry as an example structured p2p overlay protocol. It should be noted that the ideas and concepts apply equally to other protocols like Chord, CAN and Tapestry.

2. Pastry overview

In Pastry, keys and nodeIds are 128 bits in length and can be thought of as a sequence of digits in base 16. Pastry routes a message to the node whose nodeId is numerically closest to the key, in a circular nodeId space, which we call a *ring*.

Each node maintains both a leaf set and a routing table. The leaf set contains the immediate L clockwise and counter-clockwise neighboring nodes in the circular nodeId space. A node's routing table is organized into 32 rows and 16 columns. The 16 entries in row n of the routing table refer to nodes whose nodeIds share the first n digits with the present node's nodeId; the $n+1$th nodeId digit of a node in column m of row n equals m. The column in row n corresponding to the value of the $n+1$'s digits of the local node's nodeId remains empty. NodeIds are chosen randomly with uniform probability from the set of 128-bit strings. As a result, only $log_{16}N$ rows are populated in a node's routing table on average, if there are N nodes participating in the overlay. Figure 1 depicts an example routing table.

In a normal routing step, a Pastry node forwards the message to a node whose nodeId shares with the key a prefix that is at least one digit longer than the prefix that the key shares with the present node's id. If no such node is known, the message is forwarded to a node whose nodeId shares a prefix with the key as long as the current node, but is numerically closer to the key than the present node's id. Such a node must exist in the leaf set, unless all of the members in one half of the leaf set have failed concurrently. Given that nodes with adjacent nodeIds are highly unlikely to suffer correlated failures, the probability of this event can be made very small even for modest values of L. The expected number of routing hops is only $log_{16}N$. Figure 2 shows an example.

Each service is assigned a unique *service id*. When a node determines that it is numerically closest to the key (using the leaf set), it delivers the message to the local service whose service id matches that contained in the message. Moreover, the service is notified on each intermediate node that a message encounters along its route. Services use this to perform dynamic caching, to construct multicast trees, etc.

Pastry is fully self-organizing. A node join protocol ensures that a new node can initialize its leaf set and routing table, and restore all system invariants by exchanging O(log N) messages. In the event of a node failure, the invariants can likewise be restored by exchanging O(log N) messages. Like all other p2p overlays, Pastry requires a *contact node* already in the overlay to bootstrap the join protocol.

Pastry constructs the overlay network in a manner that is aware of the proximity between nodes in the underlying Internet. As a result, one can show that Pastry achieves an average delay penalty, i.e., the total delay experienced by a Pastry message relative to the delay between source and destination in the Internet, of only about two [1].

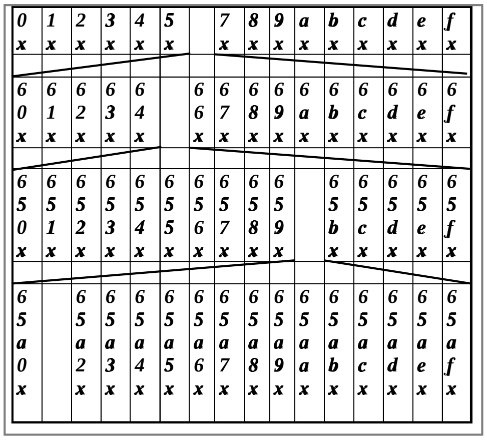
3. The universal ring

Our infrastructure for service discovery and binding relies on a *universal ring*, which is an overlay that all participating nodes are expected to join. The universal ring only provides services to bootstrap other services. Other services typically form separate overlays, which are created dynamically. The nodes in the service specific overlays are a subset of the nodes in the universal ring. The universal ring enables peers to advertise and discover services of interest, to find the code they need to run to participate in a particular service overlay, and to find a contact node to join the service overlay.

3.1. Joining the universal ring

To join the universal ring, each node needs to obtain a nodeId that is assigned by some element of a set of trusted authorities, e.g., ICANN or a certification authority like Verisign. The certification authority assigns a random nodeId to the node and signs a *nodeId certificate* that binds the nodeId with a public key for a bounded amount of time. The node knows the private key that corresponds to this public key to authenticate itself to other nodes in the overlay. The certification authority should charge nodes for the certificates it issues to make it more difficult for an attacker to control many virtual nodes in the universal ring [4].

After obtaining a nodeId certificate, a joining node needs to obtain the address of a contact node in the universal ring. If a large fraction of the nodes in the Internet are in the universal ring, one can use brute-force, distributed techniques to find a contact node. For example, expanding ring IP multicast or other forms of controlled flooding will work well because they will find a contact node within a few hops of the joining node. Otherwise, servers with well-known domain names can be used, which provide a



0x	1x	2x	3x	4x	5x		7x	8x	9x	ax	bx	cx	dx	ex	fx
60x	61x	62x	63x	64x		66x	67x	68x	69x	6ax	6bx	6cx	6dx	6ex	6fx
650x	651x	652x	653x	654x	655x	656x	657x	658x	659x		65bx	65cx	65dx	65ex	65fx
65a0x		65a2x	65a3x	65a4x	65a5x	65a6x	65a7x	65a8x	65a9x	65aax	65abx	65acx	65adx	65aex	65afx

Figure 1. Routing table of a Pastry node with nodeId $65a1x$, $b = 4$. Digits are in base 16, x represents an arbitrary suffix.

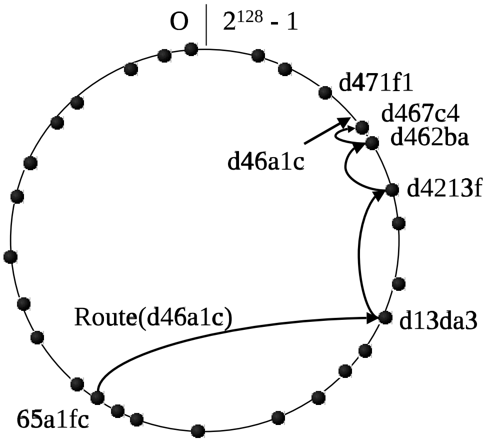


Figure 2. Routing a message from node $65a1fc$ with key $d46a1c$. The dots depict live nodes in Pastry's circular namespace.

randomly chosen contact node upon request. These techniques do not work well to find contact nodes for individual service overlays, which will likely be smaller and numerous. We describe a service that provides contact nodes for service overlays in Section 3.5.

3.2. Universal ring services

There are three basic services that the universal ring must provide to facilitate service advertisement, discovery and binding.

3.2.1 Persistent store

The first service is a persistent store for key-file pairs that provides efficient access to files given their keys. This service is used to store information about services, the code needed to run them, and lists of contact nodes for the different services. All stored files are immutable except contact lists, which do not require strong consistency semantics.

The functionality provided by the persistent store is similar to the one offered by PAST [13]. All files stored in the universal ring must be signed using a private key associated with a valid nodeId certificate.

A key-file pair is inserted in the store by using Pastry to route to the node in the universal ring whose nodeId is the numerically closest to the key. This node verifies the signature in the file and then replicates the file over the n numerically closest nodes in the ring to provide fault-tolerance against node failures.

The lookup of a file given a key also involves routing a lookup request to the node in the universal ring whose nodeId is the numerically closest to the key. The node performing the lookup then receives a copy of the signed file, which it can verify.

Most files stored in the persistent store are small and can be aggressively cached. Both on insertion and lookup these files are cached on all nodes that participate in the routing. This caching (which was also used in PAST) is important to avoid overloading nodes when there are flash crowds; it prevents the nodes responsible for storing a file associated with a service from being overloaded if the popularity of the service increases dramatically in a short period of time. Code files can be large and in this case it might make sense to cache them less aggressively.

3.2.2 Multicast

The second basic service is an application-level multicast service, called Scribe [2, 14]. Nodes wishing to subscribe to a multicast group route a request to the node whose nodeId is numerically closest to the multicast groupId, called the group's root. Each node along the path of the request implicitly subscribes to the group, and adopts the previous node along the route as a child in the group's multicast tree. The request terminates when it arrives at the root, or at a node that is already subscribed to the group. Because membership management is distributed, the system is highly scalable.

A message is multicast to the group by sending it to the root. The root then forwards the message to all its children, and so on. When a topology-aware protocol like Pastry or Tapestry is used as the underlying p2p overlay, the resulting multicast trees have the property that nodes in successively smaller subtrees are increasingly near each other in the Internet. As a result, the multicast is very efficient, both with respect to delay and link stress [2].

Moreover, using a simple and efficient search algorithm, any node in the universal ring can efficiently locate a nearby member of a given group. To find such a member, a message is routed towards the group's root. When the message reaches a subscriber, it returns its list of children. The client then contacts the nearest among these children (determined, for instance, by measuring the RTT to each child). This process continues until a a leaf in the multicast tree is reached. If the multicast tree was constructed in a topologically aware fashion, then that node is likely to be among the members that are nearest to the client issuing the search. Accuracy of this search can be traded for even higher efficiency by contacting a random child in each step; this works particularly well when members exist that are very close to the client.

The nearby subscriber search can be used to discover a nearby node in the network with a given property. Nodes with a given property subscribe to a group associated with the property. It can be used, for instance, to efficiently locate nearby nodes with certain hardware capabilities or services, nodes that have spare capacity, nodes that provide a specific service, or nodes that are operated by an organization trusted by the user.

3.2.3 Distributed search

The third basic service is a distributed search engine that allows users to find services given textual queries. Given a set of keywords and a service key, it associates the keywords with the specified key. The search engine allows nodes to search for keys using a set of query keywords. In the simple case, a boolean AND query is supported. More complex queries and ranking of query results are possible but details are beyond the scope of this paper.

We now briefly outline how the indexing could work. There are currently several projects looking at the development of searching and indexing for DHTs [5]. Here we describe a very simple scheme that can be significantly improved. The searching can be achieved by using a distributed inverted index that associates a keyword with a list of service keys. Every node in the universal ring stores part of the inverted index. The index for a keyword is stored in the node whose nodeId is numerically closest to the hash of the key. For resilience to node failure, the index on each node is replicated over the n numerically closest nodes in the ring. When a search is performed, the keywords in the query are hashed and Pastry is used to access the corresponding indices. If a keyword has an inverted index entry, the associated set of service keys is returned. The node can then take the intersection of all the sets of keys returned. The intersection represents that set of services that satisfy the query. We plan to cache results of popular queries in the path to their component keywords to prevent overloads under flash crowds as was done in the persistent store.

Persistent queries (also called triggers) can be implemented as follows. A node that issues a persistent query subscribes to a Scribe multicast group associated with each keyword that appears in the query. When a service is advertised, a notification is sent on the multicast groups associated with each of the service's keywords. The receivers intersect the notifications received on each group to which they subscribe according to the query. As an optimization, boolean AND queries can be handled by subscribing to a group associated with the conjunction of query keywords in a canonical form. The root of such a group in turn subscribes to the groups associated with each of the conjunctive term's keywords and intersects notifications in the obvious way.

In the following sections, we describe in more detail how the persistent store, the multicast service and the search engine are used to enable discovery of services, code, and contact nodes.

3.3. Service advertisement and discovery

A service is created by generating a *service certificate* that describes the service. This certificate includes the textual name of the service, a textual description of the service, and a set of *code keys* (which are described in the next section). Each code key identifies a different implementation that provides the functionality required to run the service. The service certificate is signed by the private key associated with the nodeId certificate of its creator.

To advertise a service, the creator uses the persistent store provided by the universal ring to store the service certificate reliably under a *service key*, which is equal to the hash of the certificate. The textual description of the service and the service name are then inserted by the creator into the indices of the search engine provided by the universal ring. This associates the keywords with the service key.

In order for a node to retrieve the service certificate, it must discover the service key. This is performed by keyword searching using the search engine provided by the universal ring. A user performs a keyword search to retrieve a set of service keys, and then these service keys can be used to retrieve their associated service certificates from the persistent store provided by the universal ring. Alternatively, a node interested in certain categories of new services can issue a persistent query in the search engine, in order to be notified when new services of interest are advertised.

3.4. Code binding and update

As discussed above, we allow the creator of a service to specify several acceptable implementations for the service. These implementations are not necessarily written by the service creator and they may be used by many services that provide similar functionality. Therefore, code is stored separately from service certificates.

Each implementation has a *code certificate* that includes the implementation name, a textual description of the code, and the actual code[1]. The certificate is signed by the code writer using the private key associated with its nodeId certificate in the universal ring. This signature allows users to verify that the code was written by the code writer, which is important because the user may be unwilling to run a piece of code just because the service creator vouched that it was suitable for its service.

The *code key* associated with the code certificate is obtained by hashing its contents. The persistent store provided by the universal ring is used to store the code certificate reliably under its code key.

After obtaining a service certificate, a node selects a code key, and then retrieves the code certificate associated with that key from the persistent storage service running on the universal ring.

Software updates for an implementation are inserted into the persistent store. The new code keys are then advertised on a multicast group consisting either of all members of the associated service overlay, or all nodes that use previous versions of the given implementation.

3.5. Joining a service overlay

After obtaining the service certificate and the code for a service of interest, a node is almost ready to join the service overlay. But first it needs to obtain the address of a contact node in the service overlay. We describe how to find this node next.

For each service, a small list of contact nodes is inserted in the universal ring under the service key. A node that wants to join the overlay of the service obtains this list when it looks up the service certificate in the universal ring. Then, it selects one of the nodes in the list at random to be its contact node.

To ensure that the contact list remains fresh, the oldest element in the list is replaced by the joining node. Copies of the contact list can be cached in the universal ring path to the node that stores the service key to prevent overloading this node. Additionally, each cached copy of the list can be updated independently, as described above, to ensure its freshness and to prevent overloading of the contact nodes.

P2p overlays like Pastry [12] and Tapestry [17] exploit network locality to provide better performance. They require that the contact node be close to the joining node in the underlying network topology in order to achieve this. However, because of the randomization of nodeIds it is highly likely that the contact node is not close to the joining node. This problem can be solved by performing a nearest subscriber search on a multicast group consisting of the service overlay's current members.

Alternatively, in Pastry, the problem can be solved by using the algorithm described in [1]. This algorithm uses the contact node and traverses the service overlay routing tables bottom up to find a good approximation to the service overlay node that is closest to the joining node in the network. A similar algorithm could be used with Tapestry. Once the closest node has been found, it is used to start the joining algorithm described in [1].

4. Conclusions

In this position paper, we have outlined a preliminary design of an infrastructure that provides service advertisement, discovery and binding to bootstrap services based on structured p2p overlays. This problem has not been addressed by previous work.

We have proposed the use of a universal ring that provides only bootstrap functionality while each service runs in a separate p2p overlay. The universal ring provides: an indexing service that enables users to find services of interest by supplying boolean queries; a multicast service used to distribute software updates and for coordination among members of a service overlay; a persistent store and distribution network that allows users to obtain the code needed to participate in a service's overlay; and a service to provide users with a contact node to join a service overlay. These services are self-organizing and fault-tolerant and scale to large numbers of nodes.

The solution we have proposed, whilst targeted at Pastry, is applicable to other protocols such as CAN, Chord and Tapestry. It is also applicable to service discovery and binding for traditional centralized services.

References

[1] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks, 2002. Submitted for publication.

[1] Potentially other fields could be added to code certificate, such as a documentation URL, version number, code dependency information and so forth.

[2] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, 2002. To appear.

[3] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *18th ACM Symposium on Operating Systems Principles*, Oct. 2001.

[4] J. R. Douceur. The sybil attack. In *Proceedings of IPTPS02*, Cambridge, USA, March 2002. http://www.cs.rice.edu/Conferences/IPTPS02/.

[5] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica. Complex queries in dht-based peer-to-peer networks. In *Proceedings of IPTPS02*, Cambridge, USA, March 2002. http://www.cs.rice.edu/Conferences/IPTPS02/.

[6] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A scalable peer-to-peer web cache. In *Proceedings of the 21st Symposium on Principles of Distributed Computing (PODC 2002)*, Monterrey, CA, July 2002.

[7] J. Kubiatowicz and et al. Oceanstore: An architecture for global-scale persistent store. In *Proc. ASPLOS'2000*, November 2000.

[8] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of IPTPS02*, Cambridge, USA, March 2002. http://www.cs.rice.edu/Conferences/IPTPS02/.

[9] Microsoft. Upnp specification.

[10] OMG. Corba naming service specification.

[11] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proc. of ACM SIGCOMM*, Aug. 2001.

[12] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *International Conference on Distributed Systems Platforms (Middleware)*, Nov. 2001.

[13] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th ACM Symposium on Operating Systems Principles*, Oct. 2001.

[14] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Third International Workshop on Networked Group Communications*, Nov. 2001.

[15] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[16] Sun. Jini specification.

[17] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.

[18] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-Area Data Dissemination. In *Proc. of the Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 2001)*, June 2001.