A Pretty Flexible API for Generic Peer-to-Peer Programming

Giuseppe Ciaccio
DISI, Università di Genova
via Dodecaneso 35
16146 Genova, Italy
ciaccio@disi.unige.it

DISI Technical Report DISI-TR-05-06

11 May 2005

Abstract

NEBLO is a library and runtime system based on a structured overlay network. The API presented by NEBLO offers simple primitives and powerful mechanisms for programming generic peer-to-peer applications, in a way that is independent from the underlying overlay.

The primitives allow the peers to exchange messages with one another in two different patterns, namely, unidirectional and request-response; the latter takes place in a split-phase non-blocking way, so that the application can be made latency-tolerant and thus more performing.

The semantics of messages is not defined by NEBLO itself. Rather, a mechanism is offered to the application programmer so as to allow him to write and set up application-level handlers, which are to run upon message arrivals at each peer. The overall behaviour of the application is thus shaped by the handlers, as they define the actions to be carried out when a message arrives to its recipient or passes through an intermediate peer.

The API also allows to define application-level handlers for other two typical tasks of any dynamic peer-to-peer system, namely, the migration of overlay addresses across peers after new peer arrivals, and the regeneration of missing overlay addresses after peer departures.[1]

Keywords: structured overlay, message handlers, non blocking communication, latency tolerant, peer to peer.

[1] This research is supported by the Italian FIRB project *Webminds*.

1. Introduction and motivation

Overlay networks, both structured [23, 32, 12, 30, 19, 27, 21, 35] and unstructured [8, 3], have been receiving a lot of attention by the research community as flexible and scalable low-level infrastructures for distributed applications of many kind. They have also been proposed as general networking infrastructures [15, 31, 18, 17], because of their potential ability to decouple network addresses from physical placements of cooperating hosts, an important feature for privacy and mobility.

Structured overlays are receiving far more attention lately, because of performance guarantees they can in principle provide thanks to their regular topologies. On the other hand, unstructured overlays were the foundations of the first large-scale peer to peer (p2p) applications to be deployed to a vast community of users, with an astounding success that shedded light over the immense potential of the p2p paradigm.

It is because of the success of p2p applications, that one could expect a corresponding effort towards the definition of mechanisms, abstractions, tools, and infrastructures supporting the design and implementation of p2p systems. However, it seems that such an effort has mostly been devoted to the design and validation of impressively many variants of the most popular among the low level infrastructures for p2p, namely, structured overlays. In addition to inventing more and more overlay networks, a significant effort has been put on porting to existing overlays a lot of classical distributed applications: network storage [16, 11, 28], naming [9], content publication [14, 8, 4, 26, 34, 29], multicast [24, 5], and communication security [22].

This allowed to validate beyond any doubt the generality and flexibility of overlay systems as low level distributed infrastructures.

In our opinion, it is now time for a deeper understanding of which primitives and mechanisms are better provided by the low level infrastructure in order to ease the development of distributed applications layered onto them. This indeed is the main motivation of the work presented hereafter.

2. Basic terminology

To focus our ideas, let us consider a structured ring-shaped overlay. Each peer has a *successor* in the topology. By no means is this the only possible choice; it just simplifies the exposition of our ideas, with no loss of generality.

The *overlay address space* is the set of 2^k binary words of k bits, ordered as a circle modulo 2^k. This space is mapped onto the ring of peers in consecutive chunks or *address ranges*. If peer P owns the address range $\langle A_l, A_u \rangle$, and peer Q is the successor of P, then all addresses of Q are greater than A_u (modulo 2^k).

3. Overlay events and application actions

.

3.1. Messages: send, forward, and receive

The mission of an overlay is to accomplish the exchange of *messages* among and throughout participant entities. A message directed towards the overlay address A is hopefully routed towards its *recipient*, namely, the peer whose address range contains A. The route traverses a number of intermediate peers, starting from the *sender*. Each peer on the path must take a routing decision based on local information, and according to such decision it *forwards* the message to another peer. The routing algorithm is said to converge if the message eventually reaches the recipient except in case of "accidents" (both the peers and the network links have a non-null failure probability).

A message hitting its recipient is expected to convey useful information in order for the recipient to accomplish some actions. Such actions are application dependent; for instance, in a distributed storage the action might be to store the payload of the arrived message in the local repository, thus giving the message the meaning of a write request on the provided data block.

However, a message might also be expected to trigger application-dependent actions on the intermediate peers, called the *forwarders*, which it passes through during its trip. An example is proximity caching of blocks in a distributed storage, but more examples can come to mind when considering group communication.

3.2. Unidirectional and request-response messaging patterns

Until now we have been talking about messages as individual pieces of information sent to some recipient, to trigger remote actions once there as well as along the trip. This simple model corresponds to what we call *unidirectional messaging*, in that the sender does not expect anything back as a result of having sent a message out. Instant messaging follows such a communication pattern; it is up to the recipient to decide whether to respond or not, and to do so the identity of the sender must emerge to the application level, where the answer can possibly be created and sent.

However, there are numerous cases in which a peer expects a return answer as a response to a previously sent message; the read operation in a distributed storage and the lookup operation in a DNS are two notable such cases.

It could be argued that a *request-response* communication pattern could just be implemented with unidirectional messaging, provided that the sender identity is sent along with the message and emerges at the recipient side, where the application layer may take the responsibility of sending a response back after having created it. However this is not a general solution: providing the overlay address of senders along with their messages is unwise whenever anonymity or censorship-resistance are main concerns of a distributed application.

Thus, the general solution can only be another primitive of the overlay system in the form of a specific request-response messaging feature, in addition to the unidirectional one. The runtime system of the overlay must take care of correctly returning responses back to the initiator peer, by somehow managing the return paths. One possibility is that each forwarder keeps an internal state for the in-flight requests; another, more appealing option is to keep the return path inside the messages themselves, perhaps using "onions" [25].

To tolerate the latency incurred by the response, the request-response primitive should have a split-phase, *non-blocking* architecture: sending a request and waiting/testing for the corresponding response should be distinct operations, in between which the application should be able to accomplish other useful tasks, including other communications. In many cases, accomplishing request-response cycles in a split-phase fashion pays

a lot in terms of performance. For instance, reading a file from a distributed storage could be done in a faster way by initiating a bunch of block read operations in sequence, without waiting for the completion of any of them. At the end, the total latency for the bunch is just the maximum latency of the individual blocks, rather than the sum of all latencies.

From the programming point of view, the logical link between the initiation of a request-response cycle and its corresponding waiting operation might be denoted by an opaque object called a "handle", returned as the result of the former operation and passed as parameter to the latter. This is the approach followed by MPI [1], for instance.

3.3. Different meanings for different messages

The scenario outlined so far, in which a message is to trigger an application-specific behaviour at each traversed peer and another behaviour at the final recipient, is overly simplified. Practical distributed applications are coalitions of cooperating distributed services; for instance, surfing the Internet usually requires DNS lookups for address resolution plus HTTP communications for the actual access to web pages. In the end, all these services rely on network messages; however, these messages have different formats, different meanings, and demand different treatments on the hosts they happen to reach. It is for this reason, that each host running the IP protocol is given the possibility of listening to a number of Internet ports rather than only one, and attach a possibly different daemon to each enabled port.

The concept of *communication port* actually adds nothing to the network semantics: everything could be done by using a single port, provided that messages come along with a tag, and the listening daemon is able to read the tags and take different actions depending upon these. Nevetheless, the greater abstraction level provided by ports, as opposed to message tags, provides an unquestionable degree of flexibility to the programmer of distributed applications: with separate ports, the various distributed services cooperating to the same application can be made more independent of one another.

3.4. Departures and arrivals: the dynamics of overlay addresses

In the ideal scenario in which the mapping of overlay addresses onto peers is complete, that is, no "holes" left between each peer and its successor, a message can always be routed towards its destination, at worse by traversing the successor chain starting from the sender. In a realistic scenario, however, a faulty or disconnected peer could break the successor chain and also create a "hole", a discontinuity in the address space. Routing towards backup locations and a suitable degree of application-level redundancy are not enough: the system must also quickly repair the successor chain and *regenerate* the missing addresses, or redundancy would eventually degrade.

The actual actions consequent to address regeneration depend upon the particular application on run. For instance, a distributed storage might have to rebuild lost blocks of data previously associated to the overlay addresses that have been lost after a peer departure. In general, the application instance running on a peer might have to rebuild a missing piece of its own state, possibly by using redundant copies found on other peers. Therefore, the application-level regeneration actions require a cooperation among peers, rather than taking place on just the peer who is attempting to repair the ring. Additionally, the regeneration might have to rely on some sort of redundancy of the application state.

This raises an important issue concerning how to manage redundancy of the application state. One possible approach is that the application takes care of redundancy, by implementing own policies; in this case, the cooperation among peers in case of regeneration of lost overlay addresses must be entirely programmed at application level. Another approach is that redundancy is supported by the infrastructure, at least to some extent; in such a case, the distributed task of state regeneration might be simpler to program at application level. Our opinion is that the latter approach is more general and should therefore be preferred, provided that the redundancy mechanisms present in the infrastructure could be bypassed by the application programmer if he wishes so.

Another frequent event in an overlay is the subscription by a new peer. When a newcomer joins the ring, it must be assigned an address range so that the mapping of addresses to peers remains consistent with a ring topology. This implies that some of the addresses so far owned by another peer must *migrate* to the newcomer. For performance reasons, the migration is done on a chunk basis: the peer who happens to be adjacent to the newcomer in the ring splits its own address range in two parts, then yields one part to the newcomer. As in the case of address regeneration, the actions to be carried out in case of address migration is applications-specific. For instance, a distributed storage might have to move data blocks towards the newcomer.

4. Put it all together: a general-purpose API

In this Section we propose a general-purpose API for p2p programming which provides all the basic primitives and mechanisms that have emerged after the analysis displayed in Section 3. These buiding blocks for the application layer are:

- unidirectional messaging;
- request-response non-blocking messaging;
- communication ports, to support multiple distributed services on the same infrastructure;
- receiver handlers, to give semantics to messages arriving at the recipient;
- forwarder handlers, to take actions at the intermediate peers along the path of a given message;
- migration handlers, to manage the migration of overlay addresses when new peers join the system;
- regeneration handlers, to restore the system integrity after peer departures or failures.
- optional mechanism to support some form of redundancy of the application state (Section 3.4)

The API shall be defined using a language-neutral notation similar to the one of [13]. A parameter **p** shall be denoted by →**p** if it is read-only and ↔**p** if it is read-write.

4.1. Messaging

void **UD_send** (int →**copies**, int →**port**, overlay_addr →**A**, message →**M**)

Send an unidirectional (UD) message **M** towards overlay address **A** at the specified **port**. The message is sent out in multiple **copies**, that the system will transparently dispatch towards *secondary recipients* (decided by the system itself) so as to provide a degree of redundancy.

void **REQ_send** (int →**copies**, int →**port**, overlay_addr →**A**, message ↔**M**, handle ↔**H**)

status **RES_wait** (handle →**H**)
status **RES_test** (handle →**H**)

Send out a request message **M** and wait/test for the arrival of the corresponding response. The message is sent towards overlay address **A** at the specified **port** in multiple **copies** (for redundancy again). On return from **REQ_send**() the message **M** has been registered with the system, which can reuse it to store the response if and when arrives. A value for the handle **H** is also provided, that can be used later with the **RES_wait**() and **RES_test**() to wait/test for the response arrival (or failure).

4.2. Receiver and forwarder handlers

typedef status (int →**P**, overlay_addr →**A**, message ↔**M**) **msghandler**

This is the most generic type definition of a message *handler*: a function taking as parameters all the relevant information concerning the arrived message (the destination port **P** and address **A**, and the message **M** itself). The handler can change the message; this is especially useful with request-response communications: at the recipient, the request is discarded and replaced by its response, before the system forwards it back to source. The handler returns a status value, that can be used to tell the system to suppress this message; this can be useful with forwarder handlers, in case a given message should *not* be forwarded on.

void **set_receiver_handler_UD** (int →**P**, msghandler →**rec_handler**)

void **set_forwarder_handler_UD** (int →**P**, msghandler →**forw_handler**)

Routines invoked by the application at startup time, to attach a handler to port **P** so as to manage unidirectional (UD) messages for port **P**.

The runtime system of each peer, once detected a message of unidirectional kind arrived at port **P**, evaluates whether the peer itself is recipient or not; in the former case it runs **rec_handler**, otherwise it runs the **forw_handler**.

In all cases it is the runtime system that also passes the appropriate parameters (port, address, and message) to the handler.

void **set_receiver_handler_REQ** (int →**P**, msghandler →**rec_handler**)

void **set_forwarder_handler_REQ** (int →**P**, msghandler →**forw_handler_req**)

void **set_forwarder_handler_RES** (int →**P**, msghandler →**forw_handler_res**)

Routines invoked by the application at startup time, to attach a handler to port **P** so as to manage request-response messages for port **P**.

The runtime system of each peer, once detected a *request* message arrived at port **P**, evaluates whether the peer itself is recipient or not; in the former case it runs **rec_handler**, otherwise it runs the **forw_handler_req**.

However, if the detected message is a *response* to a previously seen request, the runtime system runs the **forw_handler_res**, unless the peer is the originator of the previously seen request, in what case the message emerges to the application (see **REQ_send**(), **RES_wait**(), and **RES_test**() above).

In all cases it is the runtime system that also passes the appropriate parameters (port, address, and message) to the handler.

4.3. Migration and regeneration handlers

typedef status (overlay_addr →**A**, overlay_addr →**B**) **manager**

This is the type definition of a *manager*: a special kind of handler that runs in case of regeneration of lost overlay addresses (caused by peer departures) or migration of overlay addresses from peer to peer (due to peer arrivals). As with message handlers, the managers are application-level functions that define the behaviour of the application in the two events of regeneration and migration of overlay addresses. The relevant information in both cases is an address range $\langle \mathbf{A}, \mathbf{B} \rangle$, to be passed to the managers by the runtime system upon their invocation.

void **set_regen_handler** (manager →**M**)

void **set_migrat_handler** (manager →**M**)

Routines invoked by the application at startup time, to register a handler for, respectively, regeneration or migration of an address range of the overlay.

It is not trivial to indicate a suitable way for the runtime system to support the regeneration mechanism. Following a common-sense intuition, one could think that the regeneration handler should just run on a peer attempting to repair an adjacent "hole" in the ring. But, as pointed out in Section 3.4, the regeneration is better represented as a distributed task involving more peers, because of the need to find and use distributed redundant copies of the application state under reconstruction. Again in Section 3.4 we propose that the infrastructure offers some support to redundancy. In Section 4.1 we defined this support in terms of multiple optional copies of messages to be transparently routed towards *secondary recipient* peers. In this scenario, the application-level regeneration handler is also to run on the secondary owners of the overlay addresses under reconstruction, in that they might have copies of the lost parts of the application state. This implies that the runtime system, in case of address regeneration, sends a suitable request to the potential secondary owners of the missing addresses; this does not require any additional mechanism, because the infrastructure already "knows" how to route an arbitrary message to secondary recipients.

The regeneration handler, run on a secondary owner of a lost address, might use the communication routines of Section 4.1 to send messages aimed at recreating the lost state on the peer who is attempting to repair the ring. To this end, it is sufficient that the messages are directed towards the overlay addresses under reconstruction; the routing algorithm of the overlay will implicitly route the messages to the peer who is repairing the ring, because in doing so it has taken ownership of those addresses.

Things are simpler for the migration handler. As soon as the migrant address range has changed ownership from peer P to peer Q, the runtime system on peer P should run the migration handler. The handler might simply use the communication routines of Section 4.1 to send messages aimed at recreating the application state on Q. To this end, it is sufficient that the messages are directed towards the migrated overlay addresses; the routing algorithm of the overlay will implicitly route the messages to Q. The migration handler could also clean up the migrated information after it has been sent out.

4.4. Miscellaneous

void **init** (void)

status **subscribe** (void)

Routines invoked by the application at startup time, to respectively initialize the runtime system and join the overlay.

Typically, an application is expected to first initialize the runtime system, then register all of its handlers, and finally join the overlay so as to begin distributed cooperation by exchanging messages.

boolean **in_range** (overlay_addr →**X**, overlay_addr

→**A**, overlay_addr →**B**)

Auxiliary routine to probe the inclusion of a given overlay address **X** in the address range $\langle \mathbf{A}, \mathbf{B} \rangle$. Useful for programming regeneration handlers, a typical task of which is to pick up pieces of application state related to an address range under reconstruction.

5. Related work

The work [13] is a notable effort to provide a general-purpose API for distributed applications based on p2p structured overlays. Our work was actually inspired to that proposal, but we came up to the slightly different API accounted in this paper because in our opinion some useful features were missing and other ones were of doubtful usefulness.

For instance, the API of [13] lacks primitives for the request-response communication pattern. As pointed out in Section 3.2, such primitives are mandatory for anonymous and censorship-resistant infrastructures, and generally pay in terms of performance because of their non-blocking nature. That API also lacks mechanisms to support application-level policies for peer subscription (address migration) and departure (address regeneration).

Another point of that proposal that we do not quite agree upon, is the possibility for forwarder handlers to alter the routing and even the destination address of a passing-through message. Of course such a possibility would yield a huge degree of flexibility to the application programmer, but we feel that it would belong to the famous "90% features used by just 10% programmers". On the other hand, it forces the API itself to become heavier because of the need to refer to routing concepts (peer identities, successor lists, routing hops, secondary recipients) to be presented as opaque objects, and thus to be accompanied by a court of auxiliary functions. We feel it is better to leave the routing inside the infrastructure, and offer a more agile API to the programmer.

That said, we hope to have provided a meaningful contribution to the explicit request for feedback made by the authors of [13] in their interesting paper.

In [20], it is proposed an API for overlay programming that closely resembles the Berkeley sockets. The proposed interface is clearly simple and is founded on a widely known communication paradigm. However, in our opinion, after decades of socket programming we deserve something more powerful, flexible and higher level.

The mechanisms of message handlers are clearly inspired to Active Messages [33, 10] and to Internet ports/services in general.

6. Ongoing and future work

In the near future we plan to validate the proposed API by porting a number of classical distributed applications to it. To this end we need a working prototype of a runtime system presenting the API itself; this was one of the reasons (not the most important though) why we have recently implemented NEBLO.

6.1. NEBLO: a working prototype

NEBLO, a NEarly BLind Overlay, is a structured overlay network organized as a ring with chordal fingers [32], in which the use of imprecise finger lists, whose size and extent are severely constrained, yields a pretty good anonymity to information requestors as well as providers [7].

In the current implementation, all communications between couples of peers take place through TCP sockets, and are cyphred by a session key, exchanged in a pretty secure way (RSA) at the time of establishing the connection.

A description of the system and protocol is out of the scope of this paper; see [6] for more details.

On each running peer, a maintenance thread periodically probes liveness of the successor in the ring, so as to repair the ring itself by regenerating missing overlay addresses when a peer departure is detected. Another thread periodically rebuilds the table of long-distance routing (the fingers), so that also long-distance routing is kept up to date as peers join and leave the ring.

Request-response communication is implemented by having each request leaving a "track" on each peer traversed along its route. The track stores a unique id. for the request and a pointer to the peer which it came from. This way, it is possible to route the response back to the originator without disclosing the identity of the originator itself: it is sufficient to follow the tracks backward. The unique id. of the tracks changes at each hop in the route. However we are considering using "onions" [25] as an alternative to tracks, because with onions no state must be stored at intermediate peers, thus yielding a better resilience against certain flooding attacks.

NEBLO is presented to distributed applications in the form of a runtime library. It is free software, released under the GNU General Public Licence and available for download at [2].

References

[1] Message Passing Interface Forum, http://www.mpi-forum.org.

[2] The NEBLO homepage (in preparation), http://www.disi.unige.it/project/neblo/.

[3] K. Bennett and C. Grothoff. GAP: Practical Anonymous Networking. In *Proc. of Workshop on Privacy Enhancing Technologies (PET 2003)*, Dresden, Germany, Mar. 2003.

[4] K. Bennett, C. Grothoff, T. Horozov, and I. Patrascu. Efficient Sharing of Encrypted Data. In *Proc. of ACISP 2002*, pages 107–120. Springer-Verlag, July 2002.

[5] 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, special issue on Network Support for Multicast Communications*, 20(8), Oct. 2002.

[6] G. Ciaccio. A NEarly BLind Overlay. Technical Report in preparation, DISI, Università di Genova, 2005.

[7] G. Ciaccio. NEBLO: Anonymity in a Structured Overlay. Technical Report DISI-TR-05-05, DISI, Università di Genova, 2005.

[8] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. of Designing Privacy Enhancing Technologies: Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, July 2000.

[9] R. Cox, A. Muthitacharoen, and R. Morris. Serving DNS using a Peer-to-Peer Lookup Service. In *Proc. of the 1st International Peer To Peer Systems Workshop (IPTPS02)*, Mar. 2002.

[10] D. Culler, K. Keeton, L. Liu, A. Mainwaring, R. Martin, S. Rodriguez, K. Wright, and C. Yoshikawa. Generic Active Message Interface Specification. Technical Report white paper of the NOW Team, Computer Science Dept., U. California at Berkeley, 1994.

[11] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area Cooperative Storage with CFS. In *Proc. of 18th ACM Symp. on Operating Systems Principles*, Oct. 2001.

[12] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a dht for low latency and high throughput. In *Proc. of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, California, Mar. 2004.

[13] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, 2003.

[14] R. Dingledine, M. J. Freedman, and D. Molnar. The free haven project: Distributed anonymous storage service. In H. Federrath, editor, *Proc. of Designing Privacy Enhancing Technologies: Workshop on Design Issues in Anonymity and Unobservability*. Springer-Verlag, LNCS 2009, July 2000.

[15] J. Eriksson, M. Faloutsos, and S. Krishnamurthy. PeerNet: Pushing Peer-to-Peer Down the Stack. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, 2003.

[16] J. K. et al. Oceanstore: An Architecture for Global-scale Persistent Storage. In *Proc. of ACM ASPLOS*, Nov. 2000.

[17] M. J. Freedman and R. Morris. Tarzan: A peer-to-peer anonymizing network layer. In *Proc. of the 9th ACM Conference on Computer and Communications Security (CCS 2002)*, Washington, DC, Nov. 2002.

[18] I. Goldberg. *A Pseudonymous Communications Infrastructure for the Internet*. PhD thesis, UC Berkeley, Dec. 2000.

[19] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. Kelips: Building an Efficient and Stable P2P DHT Through Increased Memory and Background Overhead. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, 2003.

[20] J. Liebeherr, J. Wang, and G. Zhang. Programming Overlay Networks with Overlay Sockets. In *Proc. 5th Intl. Workshop on Networked Group Communications (NGC'03)*, Munich, Germany, Sept. 2003.

[21] G. S. Manku, M. Bawa, and P. Raghavan. Symphony: Distributed Hashing in a Small World. In *Proc. of the fourth USENIX Symposium on Internet Technologies and Systems (USITS'03*, Seattle, WA, Mar. 2003.

[22] P. Perlegos. DoS Defense in Structured Peer-to-Peer Networks. Technical Report UCB-CSD-04-1309, U.C. Berkeley, Mar. 2004.

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

[24] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level Multicast Using Content-addressable Networks. In *Proc. of 3rd Int.l Workshop on Networked Group Communication*, Nov. 2001.

[25] M. Reed, P. Syverson, and D. Goldschlag. Anonymous Connections and Onion Routing. *IEEE Journal on Selected Areas in Communications, special issue on Copyright and Privacy Protection*, 1998.

[26] J. Rohrer. MUTE: Simple, Anonymous File Sharing. http://mute-net.sourceforge.net/.

[27] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-peer Systems. In *Proc. of Int.l Conf. on Distributed System Platforms*, Nov. 2001.

[28] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, a Large-scale, Persistent Peer-to-peer Storage Utility. In *Proc. of 18th ACM Symp. on Operating Systems Principles*, Oct. 2001.

[29] A. Serjantov. Anonymizing censorship resistant systems. In *Proc. of the 1st International Peer To Peer Systems Workshop (IPTPS02)*, Mar. 2002.

[30] A. Serjantov. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In *Proc. of the 1st International Peer To Peer Systems Workshop (IPTPS02)*, Mar. 2002.

[31] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In *Proc. of ACM SIGCOMM'02*, Aug. 2002.

[32] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: a Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. of ACM SIGCOMM*, Aug. 2001.

[33] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, Gold Coast, Australia, May 1992.

[34] M. Waldman, A. Rubin, and L. Cranor. Publius: A robust, tamper-evident, censorship-resistant and source-anonymous web publishing system. In *Proc. of the 9th USENIX Security Symposium*, pages 59–72, Aug. 2000.

[35] 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, Apr. 2001.