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

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

Abstract

We propose and motivate an API for programming distributed applications using a structured overlay network of peers as infrastructure. The API offers simple primitives and powerful mechanisms, in a way that is independent from the underlying overlay.

The dynamic set of participants is abstracted by providing a flat space of keys, transparently scattered across all participants in the overlay. The API primitives allow application instances to send messages towards individual keys. Two different kinds of messages can be exchanged, 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 request-response pattern is also shown to be crucial for those applications demanding a degree of user anonymity.

The semantics of messages is not defined by the API itself. Rather, the API offers a mechanism to allow the application to set up handlers, which are upcalls to run upon message arrivals at each peer. The overall behaviour of the application is thus shaped by the handlers.

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 keys across peers after new peer arrivals, and the regeneration of missing keys after peer departures.

1. Introduction and motivation

Overlay networks, both structured [23, 8, 16, 19, 15] and unstructured [6, 2], have been receiving a lot of attention by the research community as flexible and scalable low-level infrastructures for distributed applications of many kinds, especially those ones based on the peer to peer (p2p) paradigm. They have also been proposed as generic networking infrastructures [11, 22, 13], 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 compared to unstructured ones, because of the performance guarantees they can in principle provide thanks to their regular topologies. On the other hand, unstructured overlays were at the core of the first large-scale p2p applications deployed to the 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 the research effort has mostly been devoted to the design, validation and evaluation of impressively many variants of 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 [12, 20], naming [7], content publication [10, 6, 24, 21], multicast [3], and secure communications [17].

All such effort allowed to validate beyond any doubt the generality and flexibility of structured overlays 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 is indeed the goal of this paper. By "distributed application" we mean any kind of pro-

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

1-4244-0910-1/07/$20.00 ©2007 IEEE

gram whose state is scattered among a set of cooperating entities, rather than kept in a centralized store. Thus, we are not addressing any specific application domain or family; the ultimate goal of a low-level p2p infrastructure is to support distributed applications in the broader sense.

The work by Dabek et al. [9] is a notable effort (the only one, to our knowledge) to provide a general-purpose API for distributed applications layered atop p2p structured overlays. Our work was actually inspired by that proposal, but we came up to a slightly different API once we tried to carefully re-think and motivate each design choice. Our API shares a lot with the API of Dabek et al., but also marks some fundamental differences; these differences shall be highlighted in Section 7.

To fix the ideas, the work presented hereafter concerns an abstraction layer that Dabek et al. call the "Key-based Routing Layer" (KRB). Again, the reader should pay attention to the fact, that we are examining primitives of a low-level infrastructure, which is supposed to be neutral with respect to any application domain.

2. Messages: name space, send, forward, and receive

Let us define the *key space* as the set of 2^k binary words of k bits. This space is (dynamically) mapped onto the (changeable) set of peers involved in the overlay, in such a way that each peer becomes *responsible* for a contiguous *key range*. The key space is a convenient abstraction for the set of peers involved in the overlay, as it abstracts from Internet addresses as well as membership changes in the overlay. The API thus provides the key space, rather than Internet addresses, as name space for the applications.

The key space is thus a name space, not a data set; each individual key always exists on its own. Specific application may associate pieces of information to individual keys, as well as leave most keys "unpopulated" (as it happens with DHTs), but this is just one particular way to use the key space provided by the low-level infrastructure.

The mission of an overlay is to allow the exchange of *messages* among, and throughout, participant entities. As we choose the key space as abstraction of the set of participants, messages are addressed towards individual keys, rather than individual peers.

A message directed towards key A must be transparently routed towards its *recipient*, namely, the peer whose key range contains A. The route traverses a number of intermediate peers, starting from the sender. At each peer on the path, the runtime system must take a routing decision based on local information concerning the neighbourhood, and according to such decision it *forwards* the message to another peer.

The routing algorithm, transparently operated by the runtime support at each participant site, 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). The convergence requirement turns into constraints on the overlay topology and the key-to-peer mapping, which must allow monotonic routing choices (each routing step must lead to a decrease of the residual distance to destination). Rings, trees, butterflies, cartesian spaces, and other topologies, each with a suitable key-to-peer mapping, are fine, but the API must abstract from the specific topology.

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 *forwarders*, namely, those intermediate peers traversed by the message 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. Two messaging patterns: unidirectional and request-response

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 uni-

directional messaging, provided that a sender identity is conveyed along with the message and emerges at the recipient side, where the application layer may take the responsibility of sending a response back. However this is not a general solution: providing sender identities along with 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.

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 allowed 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.

From the programming point of view, the logical link between the initiation of a request-response cycle and its corresponding wait/test 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 the MPI standard for message passing parallel programming [1]. An alternative architecture could be based on asynchronous notification: the initiation operation could be given the pointer to a callback routine as a parameter, and the runtime system might asynchronously run the callback when the response has successfully come back or failed.

With request-response messaging, 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 internal state for the in-flight requests; another, more appealing option is to keep the return path inside the messages themselves, perhaps using "onions" [18]. In both cases, however, a problem arises when considering that peers and their links have a non-null failure probability; what happens if an element of the return path fails before the response has travelled back? Each single request tracks a single return path, so the only way to recover from a broken return path is that the runtime system transparently *retries* the original request again, after a timeout on the missing response. In case of persistent failure after a maximum number or trials, however, the application should be returned an error. The timeout interval and the maximum number of trials are better decided by the runtime system, based on statistics of past behaviour, rather than by the application via the API. A degree of redundancy, as discussed in Section 4, may help decrease the probability of persistent response failure, by issuing more copies of the request at a time. This however is not strictly needed from the semantics standpoint (the failure probability can never be null).

4. Departures and arrivals: the dynamics of peer-to-peer

4.1. Redundancy

With a converging routing algorithm, a message would always be routed towards its destination, was the key-to-peer mapping total. In a realistic scenario, however, a faulty or disconnected peer could break the routing tables and also create a "hole", a discontinuity in the key space. The system as a whole must rely on a degree of redundancy in order to attain a minimum of availability.

This raises an important issue concerning how to manage redundancy. One possible approach is that the application takes care of redundancy, by implementing own policies. Another approach is that redundancy is supported by the runtime system, at least to some extent. The latter approach appears to be more general. Of course the application programmer must be given the possibility to bypass the runtime-level redundancy mechanism, if he thinks it is unneeded or useless or a bad match for the application.

Following the latter approach, we must require that each peer, besides being responsible for its own key range, also takes care of key ranges owned by other peers. For a given key A, we thus distinguish between its *primary copy* and the *secondary copies*. The primary copy resides at the peer who is responsible for the key range that includes A. Secondary copies, however, reside at other peers, transparently chosen by the runtime system in such a way that the routing algorithm could also converge to them. An easy way to find a place for a secondary copy of A is by applying a simple transformation to A (for instance, by toggling one of its bits), obtaining an alternative key A', and require that the peer responsible for A' also take care for A.

Each time a message is issued towards A, the runtime system should also propagate a copy of the message towards secondary copies of A. In order for recipients not to make confusion, the original key A should

also be sent along with those secondary messages. In the case of request-response messaging, issuing multiple copies of a request creates multiple return paths for responses, which increase the probability of getting at least one response back.

The presence of secondary copies of keys should be kept transparent to the applications, that is, nothing should emerge at the API level where the name space is unique. What might emerge at the API level is just the amount of desired redundancy for each given key, plus the coherency semantics established among the multiple copies. When issuing a message towards a key, each peer should be allowed to specify how many additional message replicas (if any) are to be sent towards as many secondary copies of the key, not exceeding a given system-decided maximum. The level of coherency is a controversial issue. On one hand, coherency in a distributed system is not scalable; on the other hand, different applications demand different coherency requirements. A good solution might be to have no coherency inside the runtime support, and having the API provide primitive to allow the application writer to implement own coherency policies.

4.2. Regeneration and migration of state

Implementing redundancy is not enough: the system must also quickly repair the routing tables and *regenerate* the keys that got lost after a peer departure, or redundancy would eventually degrade. The actual actions consequent to key regeneration depend upon the particular application on run. For instance, a distributed storage might have to rebuild blocks of data formerly associated to the now lost keys. In general, the regeneration of application-level state is a distributed task requiring cooperation among peers, because the peers in charge of rebuilding some lost keys must rely on redundant copies found at other peers. As the regeneration task is application-dependent, it would be the case that the API allows the application to define suitable *regeneration handlers*.

The nature of regeneration handlers is very elusive. At first glance, such handler should run at the peer, say P, who becomes responsible for the keys formerly owned by the now departed peer Q. As P is the new owner of those lost keys, P must rebuild them. P must have been notified, by the runtime system, about this new responsibility, and the notification has triggered the regeneration handler at P. Now, the handler must seek for redundant copies of the lost keys and bring them back to life. To this end, it should emit requests towards the secondary copies of the missing keys, and await responses. This is simply done by having P issue requests towards the missing keys specifying the maximum degree of redundancy, as this ensures that message replicas will travel towards (all the existing) secondary copies of keys. Once a peer R, owning a secondary copy of a key, is hit by one such request, it must emit a response which will convey back to P one of the pieces of information that P needs to restore the lost application state. The way R builds the response, however, is the real application-dependent piece of the regeneration task, whereas the way P emits the requests and awaits responses is totally generic. This makes us think that the regeneration handler is a piece of application-dependent code to be run at R rather than P. In other words, we come to a reverted scenario: P does not run any specific code for key regeneration, the runtime support at P directly emits redundant requests towards the missing keys, and those peers who happen to own secondary copies of keys, when hit by such requests, will run their own regeneration handler. These handlers will take application-dependent actions aimed at reproducing the lost pieces of state formerly associated to the primary copies of the missing keys, and will do this by emitting suitable messages towards those very keys; those messages will be routed to P and the missing pieces of application state shall be rebuilt there, thanks to the invocation of P's receiver handlers.

A more subtle point concerns regeneration of secondary copies of keys. The departure of a peer Q implies the loss of primary copies of some keys, plus the loss of secondary copies of other keys. This aspect, however, does not affect the API level, and for lack of space we must omit to discuss this point.

Another frequent event in an overlay is the subscription by a new peer. When a newcomer joins the overlay, it must be assigned a key range so that the key-to-peer mapping remains consistent with the overlay topology. This implies that some of the keys so far owned by another peer must *migrate* to the newcomer. In a ring-shaped overlay, for instance, the peer who happens to become the immediate neighbour of the newcomer should split its own key range in two parts, and yield one part to the newcomer. As in the case of key regeneration, the actions to be carried out in case of key migration is applications-specific. For instance, a distributed storage might have to flush data blocks towards the newcomer. The migration also affects secondary copies of keys.

5. Different meanings for different messages

The scenario outlined so far, in which a message is to trigger a given 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, but 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, and, if one thinks at an overlay network deployed as a generic, public networking infrastructure, communication ports become a necessary feature of whatever API for distributed applications.

6. 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 so far. The buiding blocks for the application layer are:

- unidirectional messaging;
- request-response non-blocking messaging;
- mechanisms to support some form of state redundancy;
- mechanisms to implement coherency in case of redundant state;
- 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 keys when a new peer joins the system;
- regeneration handlers, to restore the system state integrity after peer failures;
- communication ports, to support multiple distributed services on the same infrastructure.

The API shall be defined using a language-neutral notation similar to the one of [9]. A parameter **p** shall be denoted by →**p** if it is read-only and ↔**p** if it is read-write.

6.1. Messaging and redundancy

void **UD_send** (int →**copies**, int →**port**, key →**A**, message →**M**)
Send message **M** unidirectionally (UD) towards key **A** at the specified **port**. The message is sent out in as many multiple copies as specified by the **copies** parameter. The system will transparently dispatch an instance of the message towards the peer responsible for the primary copy of the key **A**, while the message replicas (if any) are routed towards secondary copies. The application writer can choose not to take advantage of this redundancy mechanism, and instead implement ad-hoc redundancy policies at application level.
void **REQ_send** (int →**copies**, int →**port**, key →**A**, message ↔**M**, handle ↔**H**)
void **RES_waitall** (handle →**H**[], status ↔**S**[])
void **RES_testall** (handle →**H**[], status ↔**S**[])
Send out a request message **M** and wait/test for the arrival of the corresponding response. The message is sent towards key **A** at the specified **port**. For (optional) redundancy, the message is sent out in as many multiple copies as specified by the **copies** parameter. On return from **REQ_send**() the message **M** has been registered with the system; the system can reuse the memory occupied by **M** to store the response if and when it arrives. A handle **H** is also provided, that can be used later with the **RES_waitall**() and **RES_testall**() to wait/test for the response arrival (or failure).

The **RES_waitall**() routine is expected to sleep, rather than busywait, until all responses related to all the handles stored in the vector **H**[] have either arrived or timed out; on return, the status vector **S**[] reports about the individual outcome of each handle (completed, timed out). The **RES_testall**() routine, however, is expected to return immediately, with status vector **S**[] reporting about the current status of each individual handle (pending, completed, timed out). In case the message has been issued in multiple copies, multiple replies might travel back to the originator peer; in this case, only one of them is selected as the "official" reply (the choice is implementation-dependent).
void **replica_keys** (key ←**R**, key →**A**, int ↔**n**)

Yield the **n**-th alternative key **R** for a given key **A**. The alternative key **R** locates the **n**-th copy of primary key **A**. This is useful when implementing coherency policies for key replicas at application level.

6.2. Receiver and forwarder handlers

typedef status (int →**P**, key →**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 key **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 can be overwritten by the corresponding response, then the system returns the response back to source. The handler returns a status value, that can be used to tell the system to suppress a given message; this can be useful with forwarder handlers, in case the 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 passes the appropriate parameters (port, key, and message) to the handler at the time of invoking them.
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 without running any handler (see **REQ_send**(), **RES_wait**(), and **RES_test**() above).

In all cases, it is the runtime system that passes the appropriate parameters (port, key, and message) to the handler at the time of invoking them.

6.3. Migration and regeneration handlers

typedef status (key →**A**, key →**B**) **manager**
This is the type definition of a *manager*: a special kind of handler that runs in case of regeneration of lost keys (caused by peer departures) or migration of keys from peer to peer (due to peer arrivals). As with message handlers, the managers are application-level functions. The relevant information for a manager is a key range $\langle \mathbf{A}, \mathbf{B} \rangle$, to be passed to the managers by the runtime system upon invocation.
void **set_refresh_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 a key range of the overlay.

The runtime system, in case of key regeneration, sends a suitable request to each potential owner of secondary copies of the missing keys. The infrastructure "knows" how to route towards secondary copies (Section 4).

The regeneration handler, run on the owner of a secondary copy of a lost key, might use the communication routines of Section 6.1 to send messages aimed at recreating the lost state on the peer P who is attempting to restore the lost information. To this end, it is sufficient that the messages are issued towards the keys under reconstruction; the routing algorithm of the overlay will implicitly route the messages to P, because P has been given ownership of those keys.

Things are simpler for the migration handler. As soon as the migrant key 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 6.1 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 keys; 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.
boolean **in_range** (key →**X**, key →**A**, key →**B**)
Auxiliary routine to probe the inclusion of a given key **X** in the key 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 a key range under reconstruction.

6.4. 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.

7. Related work

The work by Dabek et al. [9] is a notable effort to provide a general-purpose API for distributed applications based on p2p structured overlays. In our opinion, that API lacks some useful features while providing other features of doubtful usefulness.

For instance, the API of [9] lacks primitives for the request-response communication pattern. As pointed out in Section 3, such primitives are mandatory for anonymous and censorship-resistant infrastructures, and generally pay in terms of performance because of their non-blocking nature.

The API of [9] also lacks powerful mechanisms to support application-level policies for peer arrivals (key migration) and departure (key regeneration). The only available mechanism to set up a callback for arrival/departure events is the **update** (), but it is just to inform a peer that something has changed in its neighbourhood (a new peer has joined the neighbour set, or a current neighbour has departed away). As we have shown in Section 4, the regeneration of lost keys involves the owners of secondary copies of the keys, which are not necessarily neighbours of the departed peer.

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 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, neighbourhood 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 runtime support, and offer a more agile API to the programmer.

Finally, in the API proposed in [9] the support to redundancy is weak. A peer of the application may only invoke function **replicaset** () to know the names of those peers who are possibly storing secondary copies of a given key. While this allows implementing application-level redundancy and coherency policies, it causes the emersion at the API level of a new name space, namely, the name space of the individual peers. In our opinion, it is better not to explicitly name the peers at application level, because the set of peers is not static, and because censorship-resistant distributed applications may not want to explicitly bind any given piece of information to any specific peer descriptor. Our proposed API avoids such a complication by leaving peers outside the namespace, with no loss of expressiveness.

That said, we hope to have provided a meaningful contribution to the explicit request for feedback made by the authors of [9] in their interesting paper. We argue that our proposal is much simpler yet more complete, and powerful enough for p2p programming.

I3 [22] is a messaging system built on top of an overlay network. It offers messaging services, but lacks mechanisms for managing a distributed state, that must be therefore entirely built at application level, so it lacks the necessary flexibility to serve as a general middleware for distributed systems.

Meteor [14] is a middleware for distribued application more tailored to the domain of sensor networks. As such, it focuses on a content-based information management through associative matches, rather than a namespace-based content retrieval. It appears to be a higher-level middleware oriented to a specific application domain, rather than an application-neutral infrastructure.

8. 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 is one of the reasons (not the most important though) why we have recently implemented NEBLO.

NEBLO, a NEarly BLind Overlay, is a structured overlay network organized as a Chord ring [23], 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 [5].

NEBLO is presented to distributed applications in the form of a runtime library. An alternative presentation, more suitable if the system is to act as a generic networking infrastructure, would be in the form of a RPC daemon; we are considering to provide both kinds of interface. NEBLO is free software, released under the GNU General Public Licence and available for download at [4].

References

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

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

[3] 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.

[4] G. Ciaccio. The NEBLO homepage, http://www.disi.unige.it/project/neblo/.

[5] G. Ciaccio. Improving Sender Anonymity in a Structured Overlay with Imprecise Routing. In *Proc. of the 6th Workshop on Privacy Enhancing Technologies (PET 2006)*, Cambridge, UK, June 2006. Springer.

[6] 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 (PET)*, pages 46–66, July 2000.

[7] 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.

[8] 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, CA, Mar. 2004.

[9] 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.

[10] 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 (PET)*. Springer-Verlag, LNCS 2009, July 2000.

[11] 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.

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

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

[14] N. Jiang, C. Schmidt, V. Matossian, and M. Parashar. Enabling Applications in Sensor-based Pervasive Environments. In *Proc. of BaseNets 2004*, San Jose, CA, Oct. 2004.

[15] 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.

[16] P. Maymounkov and D. Mazières. 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] A. Serjantov. Anonymizing Censorship Resistant Systems. In *Proc. of the 1st International Peer To Peer Systems Workshop (IPTPS02)*, Mar. 2002.

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

[23] 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.

[24] 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.