

Rhubarb: a Tool for Developing Scalable and Secure Peer-to-Peer Applications

Adam Wierzbicki^{1,2}

e-mail: adamw@icm.edu.pl

Robert Strzelecki¹

Daniel Świerczewski¹

Mariusz Znojek¹

- 1** *Polish-Japanese Institute of
Information Technology
Chair of Parallel Systems and Networks
ul. Koszykowa 86, Warsaw, Poland*
- 2** *Warsaw University of Technology
Institute of Telecommunications
ul. Nowowiejska 15/19, Warsaw, Poland*

Abstract

Rhubarb is a platform for building peer-to-peer (P2P) applications. Rhubarb offers an API similar to Berkeley sockets. Using Rhubarb, P2P applications can be developed that are independent of centralized resources and the DNS system. Rhubarb organizes nodes in a virtual network, allowing connections across firewalls/NAT, and efficient broadcasting. The virtual network is scalable due to a hierarchical organization and efficient state management. Rhubarb is securely protected against outside and inside attacks.

1. Introduction

Peer-to-Peer (P2P) applications that utilize computers located at the edge of the Internet encounter several design limitations, such as:

- A.** no stable Internet connection or the possibility that the computer can be turned off at any time, which forces the application to be independent of centralized resources;
- B.** no DNS address;
- C.** one-way connection establishment due to NAT or a firewall;
- D.** no broadcast or multicast capability, which can be useful in many P2P applications.

Apart from the limitations described above, many P2P applications have to fulfil the following design requirements:

- I.** to be efficient in communication, and scalable to large numbers of users;
- II.** to provide a constant address to the user, who can change his IP address;
- III.** to ensure security of the user's resources and communications.

Existing P2P applications handle the described limitations in various ways (a comparison of chosen P2P applications with respect to the above criteria will be given in the conclusion of this paper), some of them using additional tools such as Dynamic DNS. Note that limitation **A** and **B** are not only a consequence of infrastructure, but of the need for resistance to lawsuits/politics. Design requirement **II** is also referred to as giving the users of a P2P application an identity. The combined design limitations **A-C** require that the application should be able to notice the presence of a user regardless of infrastructural limitations. These two notions of identity and presence are often thought of as crucial to peer-to-peer applications [9]. The enforcement of design requirement **II** is necessary to keep IP addresses private.

This paper introduces Rhubarb, a tool for building P2P applications that overcome all of the above limitations. Rhubarb attempts to fulfil all of the above design requirements, as fully as possible without using application-specific optimizations. The computers of

the users are organized into a virtual network using application-level protocols based on TCP. Rhubarb provides an API to developers of P2P applications, who can think of all their users as being connected by Rhubarb's virtual network, do not have to manage presence information and can design applications that do not need centralized resources.

While the independence from centralized resources is necessary for many peer-to-peer applications, it introduces scalability problems, since the bottleneck of distributed systems is usually communication. Rhubarb attempts to minimize communication costs and to maintain scalability. Rhubarb's efficient overlay broadcast seems to be a new contribution to P2P systems. This mechanism, along with up-to-date presence information, makes Rhubarb especially suitable for P2P games and groupware, although Rhubarb could also be used to create file-sharing applications.

In the next section, the virtual network of Rhubarb is described and it is shown how the design limitations **A-D** and **II** are taken into account. The following section discusses the efficiency and scalability of Rhubarb, which depend on the broadcasting mechanism. The next section describes the security mechanisms implemented in Rhubarb to protect it against outside and inside attacks. The last section concludes by giving a comparison of chosen P2P applications, by discussing related work, limitations of Rhubarb and directions of future work.

2. Rhubarb's Virtual Network

Addressing and state of the Virtual Network

Two types of users are present in a Rhubarb network. These are the *active* and the *passive* users. Active users agree to utilize their computational resources to help maintain the group state. Passive users do not wish to participate in synchronizing the state required for communication. They use the directory service and presence information provided by the active users. Active users maintain presence information about all users (including passive). A Rhubarb group cannot function without at least one active user, but we think that active users will be a minority in the group. The rest of this section will concern a group composed only of active users, to simplify the discussion.

The user's computers (called *nodes*) are organized to form *groups*. The state of each group is stored on each active node in that group. This state is kept up to date by a special node, called the *coordinator*. To make Rhubarb independent of centralized resources, a new coordinator can be elected among the active nodes of a group if the old coordinator fails.

Nodes are addressed based on the public keys of the user. Therefore, the address of a node is unique. However, for communications inside Rhubarb a node also has a transient session address (between joining and leaving Rhubarb). The translation of these addresses to IP addresses is part of the group state. This state includes information about every current member of the group, which consists of the members public key, IP address, port, a session address, and status (online or offline). Note that this type of addressing allows Rhubarb to fulfil design requirement **II**. The addressing based on a public key makes Rhubarb independent of the DNS system (**B**).

Group membership

Group membership can be dynamic, and nodes in a group report periodically to the coordinator. In case of a change of state (a node joins or leaves the group), the coordinator broadcasts an update of the state to the active nodes in a group. The coordinator is also responsible for allowing nodes to join the group by supplying them with the group's state and the session key of a group (see section 4.). When a new user wishes to join Rhubarb, he needs to obtain the address of a coordinator. A list of node addresses will be distributed from a *rendez-vous point* (for example, a web server). This does not violate design limitation **A**, since the rendez-vous point is used only for letting new users join Rhubarb, and can be replicated using standard techniques such as CVS. A new user (or a user who has been offline for a long time) will try to contact each node on the list to receive the address of a coordinator. From the first coordinator that he contacts, the user will obtain a list of potential groups to join with the addresses of the coordinators of these groups (the new node will join the closest group). Also, a new user can create a new group and become the coordinator of that group.

Evaluation of the prototype has demonstrated that nodes that wish to join the group should be immediately added to the group's state as offline nodes (they change state to online once they have completed the process of joining the group). In this way, in case of coordinator failure, a group will form including nodes that are in the process of joining it.

A hierarchy of groups

Rhubarb uses groups to preserve scalability by limiting the cost of synchronizing state among active nodes. Groups are completely transparent to the user. Limited group size also allows to limit the cost of local broadcast and of key management (see section 4). New groups are created dynamically, when group size exceeds a limit (for example, 100 nodes). All groups have unique addresses, which allows coordinators of each group to administer its own address space.

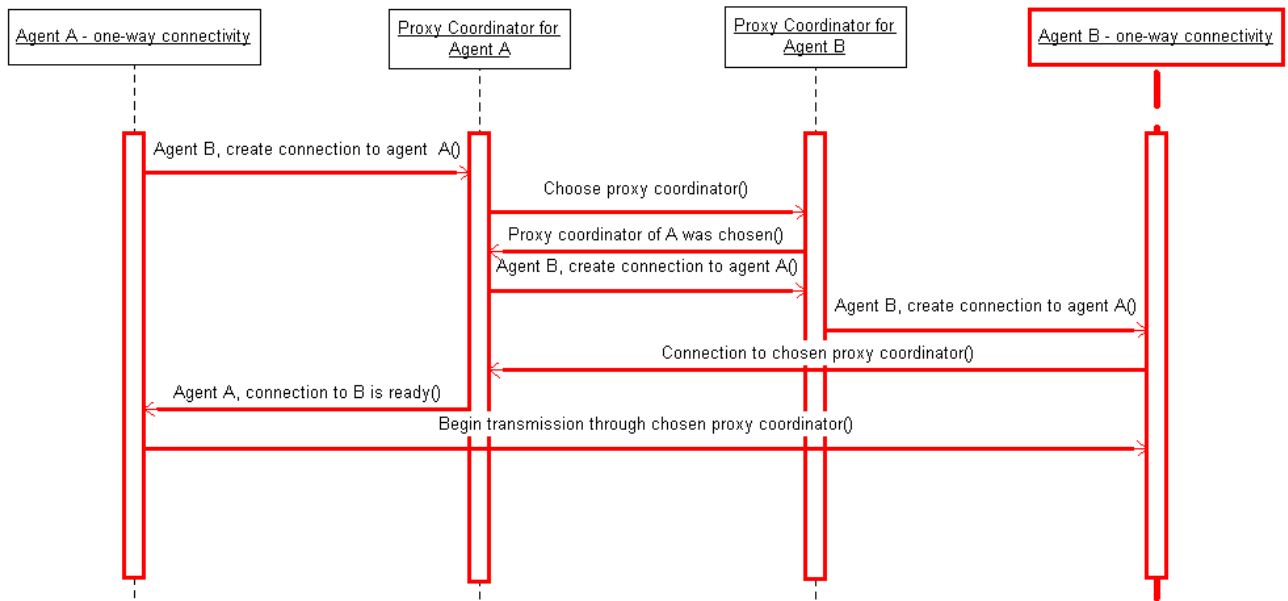


Figure 1. Communication of two agents through proxy coordinators.

Groups are organized in a hierarchy. The coordinators of first-level groups join a second-level group, and so on. When a first-level group grows beyond the maximal size, it splits into two groups. The new group elects its own coordinator. The coordinators of the two groups need to join a second-level group. If no such group exists, it will have to be created by one of the coordinators, and the second coordinator will join the new second-level group. The same scenario will occur if a highest-level group exceeds its maximal size.

Only the coordinator of a group joins a higher-level group. Therefore, if the coordinator leaves, the group will be temporarily disconnected from the hierarchy. To solve this problem, all active nodes of the group must maintain state information about the parent group, so the active node which will replace the failed coordinator will be able to join the same parent group. State updates will be propagated from the parent group to the active nodes of the child groups, but only one level down. On the other hand, state updates are never propagated up from the child groups to a parent group. Note that passive nodes can belong only to leaf groups in the hierarchy, since higher-level groups consist solely of coordinators (who are active nodes).

Presence

Presence information in Rhubarb is easy to obtain in one group, since it is part of the group state. To obtain information about the presence of a user (identified by his public key) that is not a group member, Rhubarb performs a broadcast to higher-level groups. Note that this type of broadcast does not need to reach the leaf groups, since their coordinators have up-to-date

presence information and will be reached by the broadcast in the higher-level groups. Still, this operation may be expensive. An application can improve performance by adding the IP address of a user who is present in a different group to a list of addresses who will be polled periodically to verify their presence.

Coordinator elections

If an active node finds that the coordinator has left the group, it initiates coordinator elections. When it joined the group, each active node has declared how large a group it can support as a coordinator, and this information is part of the group state. The node with the highest value wins (without any communication) and notifies the other active nodes. This mechanism minimizes the number of groups, which decreases the probability of inter-group communications (which is slower).

The use of elected coordinators and the sharing of minimal group state by all nodes makes Rhubarb independent of centralized resources (A). A P2P application that uses Rhubarb can also be independent of a central server, unless the designers of the application decide otherwise.

Information coherence

One of the main functions of the coordinator is to ensure information coherence of the system. Experience with the prototype of Rhubarb shows that one common form of incoherence is the existence of two or more separate parts of a group, who think that they are disconnected from each other. To ensure that groups are not disconnected, the coordinator of a group must try to

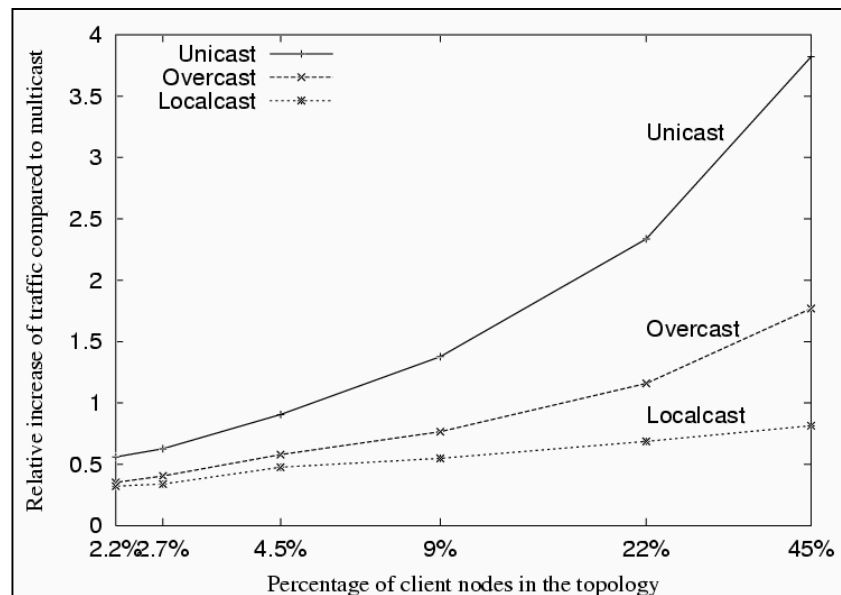


Figure 2. Efficiency of overlay broadcasting compared to multicast.

contact the remaining nodes in the group that are not currently online, unless their entry in the state has expired. Experiments with the prototype have shown that in case of loss of connectivity, a group divides into two and one part elects a new coordinator. When at a later time the communication is reestablished, one of the two coordinators will discover the other group. Then, the two coordinators need to determine (using the same rules as in an election) who will be the coordinator of the joined group.

Communicating across NAT

To facilitate communications across a firewall or NAT (C), Rhubarb uses a TCP connection to a special node outside the firewall. This node, called a *proxy coordinator*, is an active node outside a network that has one-way connection establishment. Rhubarb nodes inside the network make a permanent TCP connection to the proxy coordinator, which is renewed if it is broken by the firewall or NAT.

If a node from outside the network wishes to communicate with a node that is inside, it sends a connection request to the proxy coordinator, who forwards the request to the node inside the network. This node initiates a connection to the requesting node. If two nodes that both have one-way connection establishment wish to communicate, one of the proxy coordinators relays their communication. The communication is over two TCP connections made by both of the nodes to the chosen proxy coordinator. A detailed UML diagram of the communication between two agents through a proxy coordinator is shown on

Figure 1. Evaluation of the prototype has shown that in order to further increase efficiency, the agents inside the network with one-way connectivity can choose a representative that receives connection requests from the proxy coordinator.

Overlay broadcasting

Broadcasts in Rhubarb can have two different scopes: one group or all groups. Broadcasting to one group uses an overlay network that is constructed by a distributed algorithm. When a node joins a group, it uses that algorithm to find its position in the overlay network, which forms a tree with the coordinator at the root. The node makes its choice based on the network location of the group members, which is measured using ICMP and a limited TTL. Details of two tree construction algorithms will be discussed in the next section. Once the node has made a choice, it informs the coordinator and the chosen parent of its location in the broadcast tree of a group.

If the parent of a node leaves the group, the node must restart the procedure of finding a new position in the tree. Also, the node stops to answer its children in the tree, which is equivalent to disowning them and forces them to find a new position in the tree. In this way, the entire subtree rooted at the node that lost its parent will dissolve. If a new coordinator is elected, all nodes need to rebuild the tree before the group is fully operational.

To broadcast to all Rhubarb nodes, the message is broadcast to the higher-level groups of coordinators and then to all groups.

Virtual network management

The management of Rhubarb's virtual network is not complex. The creation of groups can be dynamic or static, if the user community wishes to adapt groups manually. Static groups can be non-transparent to applications, for example related to specific subjects of user interest. A joining node can then try to join a specific group, and if it does not find any online group member, it becomes the coordinator of the group. Apart from this option, the virtual network does not require other management tasks. The remaining tasks are related to security, which will be discussed further on.

3. Efficiency and Scalability of Rhubarb

The hierarchic organization of Rhubarb should allow scalability to millions of nodes (a three-level hierarchy of groups that have 100 nodes each is sufficient to support a million nodes). However, the synchronization of state among the active nodes in a group can use much traffic. The efficiency and scalability of Rhubarb depend on its broadcasting mechanism to one group, which is used by coordinators to synchronize state. In this section, the efficiency of the broadcast to one group will be discussed.

Recently, research has shown that overlay networks can be constructed to allow broadcasting with an efficiency that is close to multicast (for example, Overcast [1] and Yoid [2]). For Rhubarb, the traffic used to broadcast one message is the most relevant measure of efficiency. Figure 1 shows a comparison of Overcast's algorithm (modified to use hop count as a closeness measure), an algorithm called Localcast that connects every new node to the closest node in the broadcast tree in terms of hop distance, and broadcast that uses simple unicast, to the best-case efficiency of shortest-path multicast trees.

The algorithm used by Overcast to find a position of a node in the broadcast tree can be described as follows:

1. Start at the root of the tree – the root becomes a parent candidate.
2. Calculate the distance to the parent candidate
3. For each of the children of the parent candidate in the broadcast tree, calculate the distance. If the distance is not larger than to the parent candidate, the child becomes a new parent candidate and the process iterates.

Overcast has the advantage over Localcast in that it performs fewer measurements of hop distance: Localcast makes a measurement from the new node to every node currently in the broadcast tree. However, the results of our comparison show that this additional effort is justified. Note that both algorithms never create cycles, since for every new node they add only

one overlay link, thus creating an overlay graph of n nodes and $n-1$ overlay links, which must be a tree.

All measurements are averages of 50 experiments on topologies of 200 nodes (average) generated by GT-ITM to resemble closely real Internet topologies [3]. GT-ITM generates topologies in stages, by first generating the topologies of transit networks, and then topologies of stub networks that are connected to the transit networks.

On the y axis, the relative difference of the number of messages X used by an algorithm is plotted. Let M be the number of messages used by multicast, then the relative difference of X and M is given by the formula: $(X-M)/M$ (therefore, the performance of multicast is the level 0 of the Figure). The cost does not depend on the absolute size of the topology, but rather on the density of the nodes of the overlay network in the topology (proportion of user's nodes to all nodes). This value is plotted on the x axis. The result of the comparison indicate that overlay broadcasting is at worst two times as expensive as multicast, unless overlay nodes are very dense in the network.

The use of overlay broadcasting is much more efficient than using simple unicast. Note, however, that there is a tradeoff between the overall used bandwidth and the bandwidth used by individual users. For example, if all users connect over modem connections, then some of them must relay communications of others if they are parents in the broadcast tree. (If no users agree to do this, then the only alternative is direct unicast from the root.) But the parents in the tree will be forced to use their modem connections much more frequently than the nodes that are leaves in the tree.

To avoid overloading the parents, a threshold can be set on the number of children a node can have. This threshold can be set to 0 for nodes that are connected by modem connections. This modification of the algorithm can affect performance in terms of the summary consumed bandwidth, but it makes the bandwidth used by individual users more fair. To ensure that it will always be possible to create a spanning tree, active nodes are not allowed to constrain the amount of their children to zero.

The frequency of state synchronization will of course have an impact on efficiency. This value introduces a direct tradeoff between efficiency and information coherence. However, state updates only contain changes to the state, not the entire state, and are therefore rather short messages. Considering all of the mentioned optimizations, the state updates should not impose a very large network load for groups of moderate size (order of 100 active nodes).

Limitation/ Design Requirement	Rhubarb	Freenet	ICQ	Jabber	Mojo Nation	Gnutella /NG
A	+	+	-	-	+	+
B	+	+	+	-	+	+
C	+	-	-	-	+	-
D	+	-	+	+	-	+
II	+	-	+	+	+	+
III	+	-	-	-	-	-

Table 1. Comparison of chosen P2P applications

4. Security of Rhubarb

Rhubarb wants to protect the communications and resources of a user. The user is threatened by outside attacks such as sniffing, and inside attacks from other Rhubarb nodes. For protection against outside attacks, Rhubarb uses a hybrid cryptographic system: public key cryptography (RSA) for key distribution and symmetric key cryptography (3DES) for encryption. Each group has a different group session key GK for encryption, which can be changed by the coordinator. The coordinator distributes the key GK to group members by encrypting it with their public keys. That means that the broadcast mechanism cannot be used, since every node receives a different message during key distribution. However, during broadcast of an ordinary message the group key GK is used to encrypt the message, and therefore it is the same for all nodes.

While this key distribution mechanism is secure, it is not the most efficient in terms of used traffic and encryption complexity. The limitation of the size of a group is necessary to ensure that the described key distribution algorithm does not introduce scalability problems. More efficient key distribution mechanisms exist, but many of them are unsuitable for Rhubarb, since they assume that a single, central entity holds all keys used for key distribution, and no other user knows all of these keys. Rhubarb would require a distributed algorithm that distributes the key distribution keys into (possibly overlapping) sets. Each set is stored by an active node, and to change the group session key, active nodes need to cooperate. Such an algorithm has been proposed in VersaKey [13], and its integration with Rhubarb is the subject of future work.

The group key is exchanged when it expires, or when the group is compromised. To ensure that the group key is exchanged, it suffices to exchange the coordinator. Whenever a new coordinator is elected, it changes the group key.

The coordinators need to decrypt the message and encrypt it using different group keys if the broadcast is

made to all groups. When members of two different groups wish to communicate, they need to negotiate a temporary key.

Users that join the group need to authenticate their public key using a certificate or the "web-of-trust" model: a new user can be introduced to the group by a trusted user.

Inside threats can occur when one of the group members is compromised. An example is spoofing, when a Rhubarb node tries to assume the identity of another node by using the other node's public key. This is avoided by encrypting all communications of the coordinator and a node by the node's public key. A node that pretends to have a public key of another user will not be able to join the group, change its IP address, or to understand the communicator's messages.

Rhubarb's simple coordinator election mechanism can be exploited by malicious users. Since the coordinator itself can be compromised, the effects of such a situation must be limited by introducing mechanisms of control of a coordinator by other nodes. When the coordinator starts to operate incorrectly (for example, by sending incorrect data about the state of a group), the other nodes exclude it from the group and start an election. The coordinator needs to be tested repeatedly before it is excluded from the group, in order to avoid the possibility that information incoherence is taken for malicious behavior.

While Rhubarb is adequately protected, relying on automatic mechanisms alone for security is clearly insufficient. Also, when a group of users is compromised, corrective measures need to be taken, including restarting the coordinator and excluding certain nodes from the group. This is achieved by maintaining a black list as part of the group state, and allowing the list to be modified by the coordinator.

The possibility of bypassing firewalls using Rhubarb's polling mechanism can be in conflict with the security policy of an organization to which Rhubarb's users belong. For an experienced firewall administrator, noticing and blocking Rhubarb's traffic would be no

trouble at all. Therefore, users of Rhubarb should obtain permission of using their system from network administrators responsible for the security policy. Rhubarb has been developed as a tool for overcoming infrastructural limitations such as the lack of routable IP addresses for all users in an organization, which forces the organization to use NAT. It should not be considered as a tool for the theft of information.

5. Conclusion: limitations of Rhubarb and the prototype implementation

On Table 1, chosen P2P applications are evaluated with respect to the criteria described in the introduction. The evaluation was based on the available documentation of the applications [4,5,6,7,8]. For design constraints **A** and **B**, a system had to be completely independent of a centralized server or the DNS system, respectively. For **C**, it had to be possible to establish a two-way connection between two users who are both behind NAT. For **D**, any broadcast capability was considered sufficient, without reference to scalability. Design requirement **I** (scalability) is not evaluated, since it is very difficult to do this on the basis of the existing documentation. Also, Rhubarb's scalability cannot ensure that applications developed with Rhubarb will be scalable (which is discussed further on). Most systems seem to provide persistent addressing that is independent of changing IP addresses (**II**). For security (**III**), protection from both outside and inside attacks was required to consider the application to be secure.

The design requirement **I** is fulfilled by Rhubarb as well as possible, without using application-specific optimizations. For example, a file sharing application would use caching and should avoid the use of broadcast queries, using a digest mechanism such as Bloom filters to create an index of all files that is kept up to date by delta updates. This mechanism works very well for Internet caching and should improve the efficiency of file sharing applications, especially when it is used together with the presence information provided by Rhubarb to avoid routing queries to nodes that are not online. However, digest mechanisms cannot be introduced into Rhubarb since, for example, a chat application would have no use for them. Also, many game applications would require frequent broadcasting, since one user can change the state of all users in the game (the same applies to group-work applications such as whiteboards).

Several frameworks for P2P applications have been developed recently that base on the concept of a distributed hashtable [10,11,12]. In terms of communication complexity, these solutions are in between a centralized solution and a decentralized solution with completely synchronized state. They maintain a state of $O(\log n)$, where n is the number of

all users. In comparison, Rhubarb maintains $O(m)$ state, where m is the size of a group. We do not consider that the amount of memory or disk space used for the state is a concern; however, the larger the state, the more traffic will be required to synchronize it. Distributed hashtables are more efficient than the solution used by Rhubarb in a single group. On the other hand, the discussed solutions require on average $O(\log n)$ hops in the virtual network in order to route a message to any node, while Rhubarb requires $O(1)$ hops in a single group. The described solutions synchronize state lazily, and therefore cannot provide up-to-date presence information, which was one of the design constraints of Rhubarb. Also, Rhubarb achieves scalability by the use of a hierarchy and optimized broadcast for state updates, while distributed hashtables do not require hierarchies, and do not use broadcast. Considering all these factors, distributed hashtables seem better suited than Rhubarb for file sharing applications, while Rhubarb could prove superior to applications that require up-to-date presence information and efficient broadcast.

The efficiency of Rhubarb can be improved further if some of the active nodes have stable connectivity and long uptimes. The location of these nodes in the network would also be a factor of efficiency. However, Rhubarb was developed to overcome limitation **A** of P2P applications that use exclusively PCs on the edge of the Internet, and therefore could not assume that such nodes are present. On the other hand, the design of Rhubarb does not forbid such nodes, and there are ways of assuring that they will always be coordinators.

Rhubarb does not provide several levels of access control, which can be of use to some types of applications. This type of functionality is also considered application-specific. However, the search for functionality that can be considered as a "common denominator" of P2P applications should be continued and such functionality can be added to Rhubarb. Three extensions are considered at present: safe, asynchronous message delivery (when a node is down, Rhubarb will store the message and attempt to deliver it at a later time); transactions of messages (atomic delivery of sequences of messages); and extension of the broadcast mechanism to allow multicast addresses.

To summarize, applications developed with Rhubarb should be scalable, secure, fault-tolerant (if the application uses a mechanism to backup information), resistant to lawsuits/politics, and extensible. The drawback of Rhubarb could be information coherence; however, using coordinators to synchronize state is better than using a purely distributed system. Rhubarb should not be harder to administrate than current P2P systems.

Rhubarb provides an API that is similar to Berkeley sockets. It also provides a function that allows to search

for a user based on his public key (the equivalent of *gethostbyname*). Programming and porting applications for Rhubarb should not be more difficult than using SSL.

The prototype implementation of Rhubarb has been developed using Visual C++ for the Microsoft Windows platform (Windows 9x, NT, 2000). This decision was motivated by the fact that Windows is the most popular operating system for desktop computers, and the purpose of Rhubarb is to enable developers of P2P

applications access to the computational resources of PCs at the edge of the network. Initial experience with the prototype has enabled to evaluate and improve the design of Rhubarb. The experimental evaluation of the prototype's efficiency and scalability is underway, and the results of this evaluation will be the basis of further work on Rhubarb. The prototype will be provided under Gnu Public License to facilitate the development of scalable and secure P2P applications.

References

- [1] J. Jannotti, D. Gifford, K. Johnson, M. Kaashoek, J. O'Toole, "Overcast: Reliable Multicasting with an Overlay Network", Proceedings OSDI'01, 2000
- [2] P. Francis, "Yoid: Your Own Internet Distribution", Technical Report, ACIRI, April 2000, www.aciri.org/yoid
- [3] E. Zegura, K. Calvert, S. Bhattacharjee, "How to model and internetwork", Proc. IEEE Infocom, pp. 40-52, March 1996
- [4] Gnutella/ng, World Wide Web page, http://mangocats.com/annesark/gnutellang/wego_pages.html, 2000
- [5] Freenet, World Wide Web page, <http://freenetproject.org/cgi-bin/twiki/view/Main/WebHome>, 2002
- [6] Mojo Nation, World Wide Web page, <http://www.mojonation.net/>, 2000
- [7] ICQ, World Wide Web page, <http://web.icq.com/icqtour/>, 2001
- [8] Jabber Central, World Wide Web page, <http://www.jabbercentral.org/>, 2001
- [9] O'Reilly P2P Directory, World Wide Web page, http://www.openp2p.com/pub/q/p2p_category, 2001
- [10] I. Stoica, R. Morris, D. Krager, M. F. Kaashoek, H. Balakrishnan, "Chord: a scalable peer-to-peer lookup service for internet applications", Proceedings of ACM SIGCOMM'01 Conference, 2001
- [11] P. Druschel, A. Rowstron, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems", Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware'01), 2001
- [12] B. Zhao, J. Kubiatowicz, A. Joseph, "Tapestry: An infrastructure for fault-resilient wide-area location and routing", Technical Report CSD-01-1141, U.C.Berkeley, 2001
- [13] M. Waldvogel et al., "The VersaKey Framework: Versatile Group Key Management", IEEE Journal on Sel. Areas in Comm., vol. 17, no. 8, 1999