A Ring Infrastructure for Neighbor-Centric Peer-to-Peer Applications

Oliver Haase, Alfred Toth, and Jürgen Wäsch

HTWG Konstanz - University of Applied Sciences, Constance, Germany

Abstract. We propose a peer-to-peer system that supports distributed virtual world applications. For these applications, the connections between directly neighboring peers are of the utmost importance. To minimize wide area network traffic and average latency, peers that belong to the same subnet, are grouped together, and these groups are interconnected via wide area connections. To build up and maintain this optimized peer-to-peer structure, we developed a set of protocols that efficiently handle the joining and leaving of peers as well as failure situations. Peers are aranged in the logical ring structure using a two-step discovery and join procedure. The first step uses broadcast messages to discover peers in a local subnet, followed by a local join. If no peer answers in the local subnet, a remote join is performed. With the implemented recovery procedures, our peer-to-peer system can survive multi-node failures in a local subnet as well as the failure of an entire subnet.

1 Introduction

Peer-to-peer networks support a variety of different applications, including file sharing, telecommunication, multimedia streaming, web caching, distributed collaboration, and shared virtual world implementations. Evidently, a good peer-to-peer infrastructure has to efficiently support the needs of its applications [1]. For file sharing applications, e.g., the pre-dominant operation is the retrieval of a (key, value) pair for a given key. Many peer-to-peer infrastructures, such as distributed hash tables [2–6], are optimized for exactly this retrieval operation.

For distributed shared virtual world applications, however, the situation is different. Typically, the virtual world is divided into separate, neighboring areas that are distributed among the participating peers. Data exchange mainly takes place between neighboring areas, because the beings that inhabit the virtual world can move from one area to a neighboring one.

One example of a shared virtual world is Aqualife, a peer-to-peer application that simulates a distributed ecosystem. In Aqualife, each participating peer runs a part of the virtual global aquarium, and hosts fish that interact with each other, and that can swim from one peer to another. A peer has a preceeding and a succeeding neighbor that its fish can swim to and from, so that as a result all peers together form a logical ring. When a new peer joins the community, it needs to connect to a succeeding and to a preceeding neighbor, but its actual position in the ring is irrelevant from the application point of view. Thus, to

minimize network traffic and latency, it is advisable to build the peer ring upon the topological proximity of the peers.

Topological proximity takes into account not only the geographical distance between two peers, but also the characteristics of the interconnecting network including bandwidth, throughput, and latency. Evidently, determining the topological distance between any two peers is a complex and costly task; what is even more, it is a metric that varies over time, as the network load and other parameters change. Also, it is generally not possible to map the surface of the earth onto a ring while at the same time preserving topological distances.

On the other hand, the single one type of proximity that has the greatest impact on both network load and latency, is whether or not two peers belong to the same subnet. Taking that differentiation into account is very beneficial and, at the same time, feasible with comparably simple and robust procedures.

Our peer-to-peer infrastructure groups peers that belong to the same subnet together in a chain, and interconnects these local chains to a global ring, see Figure 1. This approach minimizes the amount of wide area network traffic and the average latency. In addition, it reduces the number of connections that have to traverse firewalls and NAT boxes, and that need to be taken special care of.

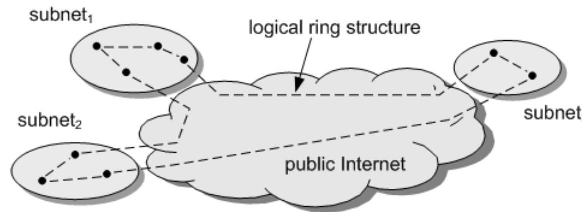

Fig. 1. Global ring structure interconnecting local peer chains.

We achieve this optimized peer-to-peer structure by a two-step discovery and join procedure. In the first step, the new peer broadcasts a discovery request into its subnet. If at least one local peer replies, the new peer initiates a *local join* procedure to have itself inserted into the local peer chain.

If no local peer responds to the discovery broadcast, the new peer performs the second step of the discovery procedure and uses a bootstrap server to be put in contact with any one peer in the community. The new peer uses the contact to request a *remote join* procedure. During this procedure, care is taken not to place the new peer between two peers that belong to the same subset, so as not to corrupt the optimal structure shown in Figure 1.

The bootstrap server is the only central entity in an otherwise serverless peer-to-peer infrastructure. It helps new peers to contact the existing community by maintaining a partial list of known peers in its peer cache. To ensure scalability, the bootstrap server caches the addresses of a constant number of peers only

and operates completely statelessly on a simple request-response protocol. Its cache replacement technique continuously updates the peer cache with new, alive contact points. This technique quickly detects inactive peers and discards them; it optimizes load balancing with respect to the join procedure; it includes even peers that have joined the community without interrogating the bootstrap server; and it diversifies the content of the peer cache across the entire peer-to-peer community [7].

2 Network Topology

Each peer a in our peer-to-peer network is identified by a globally unique identifier ID_a. We define connections between two peers a and b as follows: if a peer a knows the IP adress of peer b, there exists a (directed) connection from a to b. If both a connection from a to b and from b to a exist, we say that a connection exists *between* a and b.

From an application point of view, the peer-to-peer overlay network constitutes a logical *ring* (cf. Figure 1). This means that each peer maintains dedicated connections to its *succeeding* and *preceeding* peer in the ring. We call this type of connections *primary connections* (cf. Figure 2).

Each subnet A in the global ring has assigned a unique subnet identifier SID_A, which is known by all peers within A. The peers in a subnet that are connected to a peer in another subnet are called *edge peers*, otherwise the peers are called *inner peers* (cf. Figure 2). Subnets that are connected via their edge peers are called *neighboring subnets*.

Peers within a subnet can be totally ordered by their unique IDs. In our topology, we assume that the peers within a subnet form an ordered chain from the edge peer with the smaller ID (*lower* edge peer) to the edge peer with the greater ID (*upper* edge peer) in the subnet. Successor connections of peers point to peers with the next greater ID within the same subnet, predecessor connections point to the peers with the next smaller ID, respectively. Exception are the two edge peers in a subnet: the predecessor connection of the lower edge peer point to the upper edge peer in the preceeding network, the successor connection of the upper edge peer point to the lower edge peer in the suceeding subnet (cf. Figure 2). In case the ring is fully contained in a subnet, i.e., it is constituted solely by peers of the same network, both edge peers of the subnet are connected with each other.

To enable efficient recovery in case of failures, e.g., when a local or remote peer becomes unavailable, peers maintain so called *secondary* connections, in addition to the primary connections that constitute the application layer ring (cf. Figure 2). Each inner peer maintains 6 secondary connections: It knows both edge peers of the preceeding and the succeeding subnets, as well as the upper edge peer of the next-to-preceeding subnet and the lower edge peer of the next-to-succeeding subnet. Edge peers maintain only 4 secondary connections, since their successor (predecessor) primary connection already points to the lower (upper) edge peer of the succeeding (preceeding) subnet.

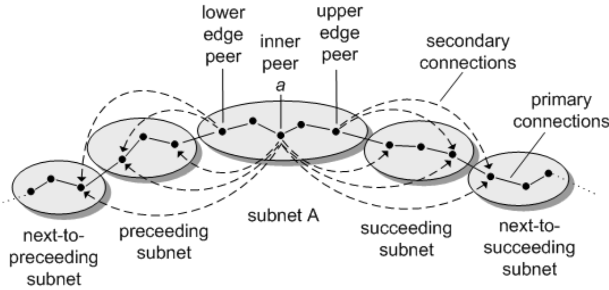

Fig. 2. Peer-to-peer network topology and terminology.

Since a peer in a subnet can efficiently determine the edge peers of its local subnet by broadcast, we do not maintain secondary connections from inner peers to the local edge peers. This decision is based upon the observation that broadcasting delivers accurate information about the local edge peers and is comparatively cheap, whereas stored information about edge peers can become out of date. This approach is not applicable to determine edge peers of neighboring subnets, since broadcasts are confined to subnet boundaries.

Please note, that, in our ring topology, it is possible for a peer to maintain connections to itself. In case the ring consists only of a single peer a, all primary and secondary connections point to a itself. Moreover, some secondary connections might be redundant, in case the ring spans less than four subnets. We allow this redundancy for the sake of a uniform treatment of the join and leave procedures and recovery.

3 Joining & Leaving

When a peer joins or leaves the peer-to-peer community, certain primary and secondary connections need to be updated. In this section we present the procedures to perform these structural updates.

3.1 Local Join

When a new peer, a, wants to join the network, it first broadcasts a `discovery request` into its local subnet. Whether or not at least one peer replies within a certain time frame, determines if a local or a remote join is performed. In this section, we assume peer a to receive at least one reply to its `discovery request`.

After the `discovery request` has been sent out, peer a waits for a time period that is long enough for a preexisting local peer to reply. Within subnet boundaries where communication is extremly fast, this period can be comparably short. In most cases, all local peers will reply within the time limit; however, for

the local join procedure to succeed, the reply of only one peer suffices, as we will see in the following.

In the reply, each responding peer sends its ID to a. Peer a selects the peer, b, with the ID closest to its own and sends a `local join request` to it. For the following considerations, let us assume ID_a to be greater than ID_b; the opposite case is treated symmetrically.

As a measure to ensure correctness even if some peers failed to respond in time to a's `discovery request`, peer b checks whether a indeed belongs between b and its current successor, c. If ID_c is less than ID_a, b rejects the `local join request` and refers a to peer c instead. Otherwise, b prepares the join procedure by sending a `lock request` to its current successor, c. If c were not locked, a concurrent join request to peer c of another new peer that also belongs between b and c could corrupt the ring structure.

Once the join lock is set in both b and c, peers a, b, and c update their primary, i.e., predecessor and successor, connections in the usual way so as to effectively insert a in between b and c. In addition, peer b transmits the subnet ID and its secondary connections to a. As a result, a knows the edge peers of the preceeding and succeeding subnets, as well as the near edge peers of the next-to-preceeding and the next-to-succeeding subnets.

If a has become a new edge peer, the secondary connections of all peers in the four neighboring (preceeding, succeeding, next-to-preceeding, next-to-succeeding) subnets need to be updated. To this end, a uses its newly aquired secondary connections to inform the near edge peers in the four neighboring subnets of the change. These edge peers use local broadcast messages to spread the information to all their local peers that update their secondary connections accordingly.

3.2 Remote Join

If the new peer, a, gets no response to its `discovery request`, it represents a new subnet, A. At this point, a's ID is made the subnet ID of A, SID_A, and will be propagated to any future peers in A, as described in section 3.1.

Peer a now contacts the bootstrap server, whose address is known through out-of-band means. The bootstrap server puts peer a into contact with a randomly selected peer, b, of another subnet, to which a sends a `remote join request`. If b, however, is an inner peer, it cannot perform the remote join itself because that would disrupt the local structure in b's subnet. Instead, b determines its local edge peers (see section 2), randomly choses one, say c, and refers peer a to c. After a has sent the `remote join request` to c, c initiates the update of the primary connections of all involved peers, i.e., a, c, and c's current remote neighbor, d. As a result, a's subnet has been inserted between two other subnets.

In the next steps, all secondary connections have to be established and updated, respectively. This is done by information propagation through the four neighboring subnets of peer a, as sketched in Figure 3. The direct neigbors of a broadcast the existence of a into their local subnets (1). As a result, all peers in

the directly neighboring subnets can update their secondary connections accordingly. In step (2), the far edge peers in these subnets send a notification back to a enabling it to set its preceeding and succeeding far edge peer connections. Also, these far edge peers propagate the update to their direct neighbor in the next-to-preceeding and next-to-succeeding subnets (3). The edge peers in these subnets broadcast the update to their local peers (4a), and send a notification back to a (4b) which sets its final two secondary connections. Please note that steps (4a) and (4b) can be performed concurrently.

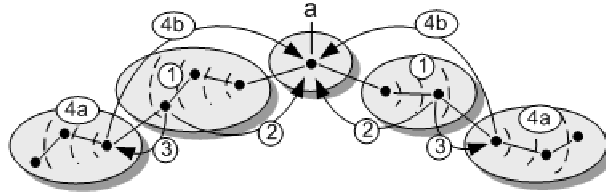

Fig. 3. Propagation of secondary connection information after remote join.

3.3 Leave

When a peer, a, wants to leave the peer-to-peer community, it first sends lock messages to its predecessor, b, and its successor, c. As a result, b and c cannot perform any other join or leave operation before a's leave has been completed, thus avoiding inconsistencies stemming from concurrent operations affecting the connections of the same peers. In the next step, a sends references to each other to b and c, which update their primary connections accordingly. Finally, a unlocks b and c, enabling them to accept future join and leave requests.

If a was an edge peer, the secondary connections of the peers in the four neighboring subnets need to be updated as well. The updated information is propagated by means of local broadcast messages and handed-off from the direct neighboring subnets to the next-to-direct neighboring subnets by point-to-point message exchange between the respective edge peers. This process is very similar to the secondary connection update in the case of a remote join operation as explained in section 3.2.

4 Recovery

In case one or more peers fail (due to hardware or software failures) or become disconnected from the network (due to network failures), recovery procedures must take place to rebuild the ring structure.

When a peer detects a failure, it initates the appropriate recovery procedure. Considering our ring topology, we can distinguish three different kinds of failures.

First, one or more inner peer fail within a subnet. Second, an edge peer fails within a subnet. Third, a subnet as a whole becomes unavailable.

4.1 Inner Peer Failure

If an inner peer fails, recovery can be handled locally within the subnet the failed peer belongs to. The peer a that detected that its preceeding (succeeding) peer failed, simply sends a broadcast message in the subnet stating that peer a looks for a new predecessor (successor). All alive peers that miss a successor (or predecessor) answer upon this broadcast with their unique peer ID and their IP address. Peer a re-connects with peer b, where $|\mathrm{ID}_a - \mathrm{ID}_b|$ is minimal. Since the peers within a subnet are totally ordered by their unique peer IDs, a re-connects with the the correct peer to close the ordered chain in the subnet and, thus, close the ring. Simultanous failures of neighboring inner peers are handled with this approach like the failure of a single peer.

4.2 Edge Peer Failure

If an edge peer e fails, the peer a that detects the failure determines if the failed edge peer belongs to the same subnet as a or to a neighboring subnet. In the former case, a uses its secondary connection to the upper (lower) edge peer of the preceeding (succeeding) subnet to re-connect to this peer and to close the ring. At the same time, peer a becomes a new edge peer of its subnet.

In the latter case, i.e., if the failed edge peer e belongs to a different subnet, peer a is itself an edge peer. Peer a uses its secondary connection to the lower (upper) edge peer b of the preceeding (succeeding) subnet to get a "handle" to the subnet S where e belongs to. The edge peer b initiates a local search for the new upper (lower) edge peer in S and returns this information to a. Peer a re-connects to this peer to close the ring. In both cases, recovery results in updates of secondary connections which is done analogously to join and leave (cf. section 3).

4.3 Subnet Failure

The case that a complete subnet gets unavailable, can not directly be recognized. The situation that both edge peers of a subnet can not be reached does not imply that all peers in the subnet became unavailable. We have to wait for a specific period of time in which we allow the subnet to recover from local edge peer failures to close the ring again. If this does not happen, the edge peer that detected the failure of the subnet uses its secondary connection to the upper (lower) edge peer of the next-to-preceeding (next-to-succeeding) subnet (cf. Figure 2) to reconnect to this edge peer and, hence, to close the ring again.

Summarizing, using the described recovery procedures, our peer-to-peer ring infrastructure can survive failures of one or more inner peers, one or both edge peers of a subnet and an entire subnet. Of course, there exist situations where

the peer-to-peer ring system can not recover from a failure: if two or more neighboring subnets fail, our system can not recover from this situation

In this case, there exist several possibilities to react and to bring the remainder of the peer-to-peer system again into a consistent state. For example, all peers can simply terminate. If configured, they can restart afterwards and build up a new, differently structured, ring. An alternative is to enable a global ring search for the "open ends" of the ring followed by re-connecting the "open ends" and, hence, closing the ring. In the current implementation, the system terminates and restarts again.

More details on recovery in our peer-to-peer ring system and its implementation can be found in [7].

5 Implementation

The presented protocols and mechanisms have been fully implemented in Java [7]. All broadcast messages use datagram sockets for UDP/IP broadcast within subnet boundaries. All point-to-point communication uses the Java Remote Method Invocation (RMI) mechanism which exchanges request/response message pairs over TCP/IP; this comprises all messages for joins (both local and remote), leaves, heart-beating, propagation of secondary connection information, and last not least the exchange of application level payloads.

Due to Java's platform independence, the software should run on any machine that provides a Java 5 runtime environment, or higher. However, the otherwise fully portable system uses one platform specific system call to access the MAC (media access control) address of a peer machine. This system call has been tested on Windows, Linux, Solaris, and MacOS, and should also work on any other UNIX based operating system.

For deployment, we use Java Web Start technology. A Web Start enabled Java application can be dynamically downloaded from a web page, similar to an Java applet except that a Web Start application runs outside the protected applet sandbox and, thus, is allowed to communicate with other peers in the network. Also, using Web Start ensures that each client always runs the latest version of the software, which dramatically simplifies deployment and versioning.

6 Related Work

The topologies of most distributed hash table approaches seem, at first glance, similar to our approach. They use logical ring structures to divide the address space up amongst the peers, and they use some notion of routing tables, finger tables, or neighbor lists to keep connections to a number of other peers in the overlay network, which resembles some similarity with the secondary connections in our approach [2, 4–6]. In the case of Chord [2], e.g., each peer maintains a so called finger table that consists of references to the peer opposite to itself in the ring, to the peer a quarter rotation ahead, to the peer an eighth rotation ahead, and so on. As a result, with the need to store (and keep up-to-date) $O(\log n)$

connections, an arbitrary peer can be reached in $O(\log n)$ steps in a Chord ring of n peers, performing a distributed binary search. Other distributed hash table approaches vary in the technical details, but follow the same principle.

A peer being able to efficiently reach any other peer in the network is essential for a distributed hash table, because the storage and retrieval of (key, value) pairs are the primary operations. From the application point of view, the peer-to-peer community is an interconnected set of data stores that cooperate to jointly provide a uniform address space. When a peer is asked for a specific key, it is mandatory that it may help to efficiently retrieve the (key, value) pair independent of where it is hosted in the overlay network. In a distributed hash table, the connections with the adjacent neighbors are of no greater relevance than the connections with any other peers. Putting physically close peers into topological proximity in the peer-to-peer overlay network, is not helpful and, thus, not a design goal.

This is in sharp contrast to our approach that is tailored for virtual world applications that pose significantly different requirements upon the underlying peer-to-peer infrastructure [8, 9]. Here, what the application sees is exactly the ring topology, and thus the connections with the adjacent neighbors are of the utmost importance. All application level data exchange goes through these connections only. Consequently, grouping physically close peers together is a highly desirable design goal for an infrastructure of this type.

Finally, the routing structures in a distributed hash table, such as the Chord finger tables, contain suitable subsets of *primary* connections, as they are used during regular operations. Our secondary connections between non-adjacent peers, in contrast, are required to ensure robustness and reliability and are only used to restore operability in case of a system or network failure.

7 Conclusion & Future Work

We have presented operations and protocols that establish and maintain a ring infrastructure for neighbor-centric peer-to-peer applications. Even though the primary topology is simple, the need for robustness and failure tolerance in a volatile and vulnerable environment such as the public Internet, makes the structures and procedures rather complex. In particular, we use different categories of secondary connections to protect the peer-to-peer overlay network against various degrees of failures.

A next step is to extent the ring to a grid structure that spans a virtual globe, with each peer having four neighbor, rather than two. Clearly, the challenge with this extension is to find a good trade-off between complexity stemming from the secondary connections and protection against multiple simultaneous node and network failures.

Other interesting conclusions relate to the Java RMI implementation rather than the protocols themselves. For one, the RMI built-in detection of a failure in the remote node is far too slow in a mixed Linux/Windows environment. We had to implement an application layer time-out mechanism to overcome this

issue. An even more serious problem is communication between two peers with a firewall in the middle. The standard Java RMI solution to this problem is to tunnel RMI requests over HTTP. This, however, requires a peer that sits behind a firewall either to run an HTTP server, or the external HTTP server in the subnet to forward incoming RMI-over-HTTP requests via a specific CGI script to the target peer. Both options are rather heavy-weight and are likely to conflict with the security policies in the subnet.

Therefore, many peer-to-peer networks employ an approach where a peer behind a firewall actively contacts a rendez-vous peer outside the firewall to establish contact with its outside neighbors. Then, TCP connections are kept open to the direct neighbors over which data can then be exchanged as desired. However, because Java RMI uses a simple request response protocol, the lifespan of a TCP connection is left to the operating system and cannot be controlled by the peer-to-peer infrastructure. In a follow-up project we plan to extend Java RMI in a way that better suits the need of a peer-to-peer infrastructure.

References

1. Hauswirth, M., Dustdar, S.: Peer-to-Peer: Foundations and Architecture (in German). Datenbank Spektrum **5**(13) (May 2005) 5–13
2. Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In: Proceedings of ACM SIGCOMM 2001, San Diego, CA (September 2001) 149–160
3. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Schenker, S.: A scalable content-addressable network. In: Proceedings of ACM SIGCOMM 2001, San Diego, CA (October 2001) 161–172
4. Zhao, B., Huang, L., Stribling, J., Rhea, S., Joseph, A., Kubiatowicz, J.: Tapestry: A resilient global-scale overlay for service deployment. IEEE Journal on Selected Areas in Communications **22**(1) (January 2003) 41–53 Special Issue on Service Overlay Networks.
5. Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Middleware. Volume 2218 of Lecture Notes in Computer Science., Heidelberg, Germany, Springer (2001) 329–350
6. Aberer, K.: P-Grid: A Self-Organizing Access Structure for P2P Information Systems. In: Proceedings of CoopIS. Volume 2172 of Lecture Notes in Computer Science., Springer (2001) 179–194
7. Toth, A.: Ein ringbasiertes Peer-to-Peer-System auf Basis Java RMI. HTWG Konstanz - University of Applied Sciences (January 2007) Diplomarbeit.
8. O'Connell, K., Dinneen, T., Collins, S., Tangney, B., Harris, N., Cahill, V.: Techniques for handling scale and distribution in virtual worlds. In: Proceedings of the 7th ACM SIGOPS European Workshop, ACM SIGOPS (1996) 17–24
9. Walch, J., Steggles, P.: User intent as a bandwidth conservation heuristic in shared virtual environments. In: IMSA. (2002) 5–10