A Peer-To-Peer System for Virtual World Applications

Oliver Haase
Alfred Toth
Jürgen Wäsch

HTWG Konstanz, University of Applied Sciences, Computer Science Department
haase|alfred.toth|waesch@htwg-konstanz.de

Abstract

Peer-to-peer infrastructures support a variety of applications, but are mostly optimized for file sharing. In this paper, we propose a peer-to-peer system that especially supports distributed virtual world applications. For this category of 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. Derived from the network topology, peers are arranged 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 has to be performed. This second step involves a bootstrap server which is used to discover peers in remote subnets. 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 [2]. For file sharing applications, e.g., the predominant operation is the retrieval of a (key, value) pair for a given key. Many peer-to-peer infrastructures, such as distributed hash tables [3, 4, 7, 5, 1], 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 properties, because the beings that inhabit the virtual world can move from one property 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 preceding and a succeeding neighbor that its fish can swim to and fro, 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 preceding neighbor, but its actual position in the ring is irrelevant from the application point of view. Thus, to minimize network traffic and latency, it seems 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, 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, at the same time feasible with comparably simple and thus 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.

We achieve this optimized peer-to-peer structure by a two-step discovery and join procedure that a newly arrived peer performs in order to join the infrastructure.

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 subnet, 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 make contact with 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 include 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 [6].

The remainder of the paper is organized as follows: In section 2 we introduce the topology of our peer-to-peer overlay network, consisting of the primary connections to the direct neighbors and secondary connections to further away peers that help to recover from various degrees of network and node failures. Section 3 describes the join and leave procedures and how they keep intact the primary and secondary connections of the peer-to-peer infrastructure. In section 4 monitoring and recovery are discussed, and in section 5 the key implementation aspects are summarized. Section 6 finally concludes the paper.

2 Network Topology

Each peer $a$ in our peer-to-peer network is identified by a globally unique identifier $ID_a$. We define con-

\footnote{The MAC address of the network adapter, which is globally unique, is sufficient for our purposes, since we assume only a single peer per machine. If more than one peer is allowed per machine, an additional component, e.g., a timestamp, has...}
connections between two peers $a$ and $b$ as follows: if a peer $a$ knows the IP address 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 preceding 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 $\text{SID}_A$, which is known by all peers within $A$. The peers in a subnet $A$ that are connected to a peer in another subnet $B$ 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 ID. 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 points to the upper edge peer in the preceeding network, the successor connection of the upper edge peer points to the lower edge peer in the succeeding 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.

In the remainder of the paper, we assume that the data structure for a connection from peer $a$ to peer $b$ looks like follows: $[\text{IP-Address}_b, \text{ID}_b, \text{SID}_B]$, where $\text{SID}_B$ is the ID of the subnet $b$ belongs to.

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, additionally to its 2 primary connections: it knows both edge peers of the preceeding and the succeeding subnets, and it knows 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.

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

Joins and leaves are the two operations that change the network structure. A change of the topology generally affects a certain number of primary and secondary connections that 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, its first action is to broadcast a discovery request into its local subnet. Whether or not at least one existing peer replies within a certain time frame or not, determines if a local or a remote join is performed. In this section, we assume that peer \( a \) receives 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 extremely 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 not, i.e., 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 acquired 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 peer that update their secondary connections accordingly.

3.2 Remote Join

If the new peer, \( a \), gets no response to its discovery request within a certain time period, it assumes to be the first peer in 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\(^2\). 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, see left side of Figure 3. 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, as shown on the right side of Figure 3.

\(^2\)In the case of Aqualife, the address of the bootstrap server is hardcoded in the application itself. This approach is feasible because Aqualife uses the Java Webstart mechanism for deployment, which means that a change of the bootstrap server address is automatically propagated into each Aqualife installation.
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 4: The direct neighbors 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 connects. Please note that steps (4a) and (4b) can be performed in parallel.

### 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.

#### 4.1 Detection and Classification of Failures

The first step to achieve failure-resilience is to detect a failure. In our system, there exist different cases: First, a peer $a$ within the distributed virtual world application might detect a failure of a neighboring peer $b$ when it tries to exchange application data with $b$. Second, a peer $a$ might detect the failure of neighboring peer $b$ when it tries to communicate with $b$ during a join or leave. Third, in our system, each peer continuously monitors its two neighboring peers: each peer sends periodic heartbeat messages to its neighboring peers; if a peer $a$ does not receive a heartbeat from a neighboring peer $b$ within a specific period of time, peer $a$ assumes that peer $b$ became unavailable. In all these three cases, the range of failure detection per peer comprises its preceeding and
succeeding peer.

When a peer detects a failure, it initiates 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.

4.2 Recovery Procedures

If an inner peer fails, recovery can be handled locally within the subnet the failed peer belongs to. The peer that detected that its preceeding (succeeding) peer failed, simply sends a broadcast message in the subnet stating that peer looks for a new predecessor (successor). All alive peers that miss a successor (or predecessor, resp.) answer upon this broadcast with their unique peer ID and their IP address. Peer a re-connects with peer b, where |IDₐ−IDₜ| 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. Simultaneous failures of neighboring inner peers are handled with this approach like the failure of a single peer.

If an edge peer or a complete subnet fails, the (primary) connection between subnets is interrupted. In this case, recovery can not be handled locally within a subnet, since broadcasts are technically restricted to subnet boundaries. Instead of broadcasts, the primary connections together with the secondary connections, i.e., redundant information about the current network, has to be used for recovery purposes.

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). The special case that the ring comprises only a single subnet and one of its "artificial" edge peer fails, is handled similarly to local recovery using broadcast to find the new edge peer and to close the ring accordingly.

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

---

3 Of course, it is possible to maintain more secondary connections, i.e., to edge peers of farther subnets. This improves the existing fault-tolerance of the peer-to-peer ring, but makes the joining, leaving, and recovery procedures more complex. We have opted for 4-6 secondary connections, since we experienced a good tradeoff between fault-tolerance and performance. This situation is similar to commit protocols in distributed database systems: 3-Phase-Commit is more fault-tolerant than 2-Phase-Commit, but more complex and expensive—in practice 2-Phase-Commit is used.
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 [6].

5 Implementation

The protocols and mechanisms presented in the preceeding sections have been fully implemented in Java [6]. 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 webpage, similar to an 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 Conclusion & Future Work

Virtual world applications pose requirements upon the underlying peer-to-peer infrastructure that vary significantly from those imposed by file sharing applications. In particular, in a virtual world the connections with the direct neighbors are of the utmost importance. However, secondary connections between non-neighboring peers are required to ensure robustness and reliability in the case of node and network failures. It is the necessity to keep these secondary connections up-to-date that makes the infrastructure and its operations complex, considering the simple topology of the ring structure.

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 will be 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


