Hindawi Publishing Corporation
International Journal of Digital Multimedia Broadcasting
Volume 2010, Article ID 470813, 15 pages
doi:10.1155/2010/470813

Research Article

RTSP-based Mobile Peer-to-Peer Streaming System

Jani Peltotalo,[1] Jarmo Harju,[1] Lassi Väätämöinen,[1] Imed Bouazizi,[2] and Igor D. D. Curcio[2]

[1] *Department of Communications Engineering, Tampere University of Technology, P.O. Box 553, 33101 Tampere, Finland*
[2] *Nokia Research Center, P.O. Box 1000, 33721 Tampere, Finland*

Correspondence should be addressed to Jani Peltotalo, jani.peltotalo@tut.fi

Received 1 June 2009; Revised 12 November 2009; Accepted 6 January 2010

Academic Editor: John Buford

Peer-to-peer is emerging as a potentially disruptive technology for content distribution in the mobile Internet. In addition to the already well-known peer-to-peer file sharing, real-time peer-to-peer streaming is gaining popularity. This paper presents an effective real-time peer-to-peer streaming system for the mobile environment. The basis for the system is a scalable overlay network which groups peer into clusters according to their proximity using RTT values between peers as a criteria for the cluster selection. The actual media delivery in the system is implemented using the partial RTP stream concept: the original RTP sessions related to a media delivery are split into a number of so-called partial streams according to a predefined set of parameters in such a way that it allows low-complexity reassembly of the original media session in real-time at the receiving end. Partial streams also help in utilizing the upload capacity with finer granularity than just per one original stream. This is beneficial in mobile environments where bandwidth can be scarce.

1. Introduction

Peer-to-Peer (P2P) streaming applications are gaining more and more users around the world. These applications allow end-users to broadcast content throughout the Internet in real-time without the need for any special infrastructure, since the user's device, together with all other peers, collectively forms the infrastructure. Furthermore, dedicated servers are no longer required since every peer can serve data to other peers. This is in contrast to a service like YouTube [1] which still requires content to be uploaded to a central server first. Some of the currently existing P2P streaming applications, such as Octoshape [2] and SopCast [3], are suitable to be used in a mobile environment but still there are many issues to be solved before an optimized solution for mobile devices can be realized [4].

With real-time P2P streaming there is no need to download the entire media file before playback can be started. Decoding can be started as soon as enough data is buffered in the peer. This avoids long startup times, and eliminates the need to store the entire content on the mobile device which still has a relatively small amount of internal memory compared to the increasing size of the actual media. In live streaming, video of an ongoing event, like a football match, is delivered as a stream in real-time. After an initial buffering period, the user starts to watch the stream from a certain location and all peers consume data in the same time window. With a Video-on-Demand (VoD) streaming the user searches a video from some catalogue, and after a certain amount of initial buffering the user starts to play the video from the beginning.

In order to increase the robustness and to accommodate the limited up- and download bandwidth between peers in the network, the original multimedia session needs to be split into smaller parts, which can be reassembled at the receiving peers into the original media representation. This paper presents an effective real-time P2P streaming system where original Real-time Transport Protocol (RTP) [5] sessions related to a media delivery are split into a number of so-called partial streams according to a predefined set of parameters. This approach allows low-complexity reassembly of the original media session in real-time at the receiving end.

The structure of the remainder of this paper is as follows. The related work is discussed in Section 2. Then, a short

overview of the system is given in Section 3. Detailed descriptions of the overlay network and the media delivery are given in Sections 4–8. After that, results from the performance experiments are presented in Section 9. Interesting areas for further work are discussed and highlighted in Section 10. Finally, Section 11 concludes this paper.

2. Related Work

Many P2P file sharing applications make use of multiple source distribution. A file is first partitioned into pieces or chunks, typically of equal size. A peer then connects to the seeder or leecher peers, and requests the missing pieces of the file in a random order. The difference between a seeder and a leecher peer is that the former has a complete copy of the file while the latter has only a partial copy. For example, with BitTorrent [6] the complete multimedia file can be partitioned into blocks of 256 KB which are then selected by the interested peers and requested according to a rarest-first piece selection algorithm. This approach is not at all suitable for streaming applications as it does not consider the delay problem. Users may experience long download delays of possibly several days. It also assumes that the full content is known and available at the source peers, which does not necessarily apply to streaming applications as the stream may be live.

A P2P multimedia streaming solution based on the BitTorrent protocol is proposed in [7]. The rarest-first chunk downloading policy is replaced by a policy where peers first download chunks that will be consumed in the near future. The tit-for-tat peer selection policy is also modified to allow free tries to a larger number of peers to let peers participate sooner in the multimedia distribution. Another P2P streaming system based on a P2P file sharing implementation was proposed already in [8]. However, the data partitioning based on fixed byte ranges is not suitable for streaming a continuous media, which is of variable bit rate nature.

In P2P content distribution, an overlay network is created at the application layer in order to transfer the actual content among peers in the network. A random mesh-based overlay architecture, like in [9, 10], provides flexibility for handling peer departures, but good general connectivity between peers is not usually achieved. There have been many studies about how to organize peers in an efficient and scalable way. In [11] receivers are organized into a hierarchy of bounded-size clusters and the multicast tree is built based on that. In [12] peers are organized into a directed acyclic graph to enable peers to obtain locality awareness in a distributed fashion. To improve the file sharing performance of the BitTorrent protocol, an overlay network where peers are grouped into clusters according to their proximity is proposed in [13]. Even though some solutions have proven their functionality with wired connections, those might not be suitable for the mobile environment.

Preliminary results of the mobile P2P system described in this paper have been published in [14]. In the following sections more information about the system is given by explaining in detail the extended Real Time Streaming Protocol (RTSP) [15] messages used for signalling and providing more results from the experiments.

3. General System Overview

The architecture of the system is designed to be scalable and efficient for real-time streaming services in the mobile environment. The system supports both live and VoD streaming services. Location awareness in terms of peer proximity has been exploited to reduce delay, and thus, to improve the scalability of the system.

Peers are grouped into clusters according to their proximity in order to efficiently exchange data between peers. For VoD streaming services, the clusters could be constructed for example, based on the interest level for certain pieces of data, so that the peers watching the same part of a video at the same time belong to the same cluster. In live streaming services a cluster can be formed only based on the proximity of peers, because all peers are interested in the same data pieces within the same time window. Clusters will also help with scalability issues of peer maintenance. Peers inside a cluster are considered to be close to each other and thus communication between peers can be done more efficiently.

All overlay network operations in the system are implemented using extended RTSP messages. All RTSP methods are extended to include an additional `RTP2P-v1` tag in the `Require` header field. This tag makes it possible for the receiving peer to detect that support for the real-time P2P extensions is needed. Additionally, all RTSP messages will include a `Peer-Id` header field to indicate the source of a message. The most important new header field is called `overlay` and it is used widely in the overlay network operations. The usage of the `Overlay` header field and other additional header fields depending on the message type are explained in Sections 4 and 7. The syntaxes of the `Peer-Id` and `Overlay` header fields in Augmented Backus-Naur Form (ABNF) [16] are given below:

```
Peer-Id = "Peer-Id:" SP id CRLF id = 1*DIGIT

Overlay = "Overlay:" SP operation CRLF
operation = "backup" | "create" | "join_bcl" |
            "join_neighbor" |"join_peer" | "leave" |
            "new_peer_id" | "remove" | "split" |
            "update"
```

The RTSP Uniform Resource Locator (URL) is formatted according to the ABNF syntax shown below. The `host` and `port` parts are defined in [17, Section 3.2]. The `service-id` specifies the service and the `stream-id` specifies the RTP session. Like in [15], it is also possible to use the asterisk character instead of the URL meaning that the request does not apply to any particular resource:

```
rtsp_URL = "rtsp://" host [":" port] ["/" [ service-id
           ["/" stream-id]]]
service-id = 1*DIGIT
stream-id = 1*DIGIT
```

Peers exchange actual media data between each other using RTP. The system is using time-based chunking, which

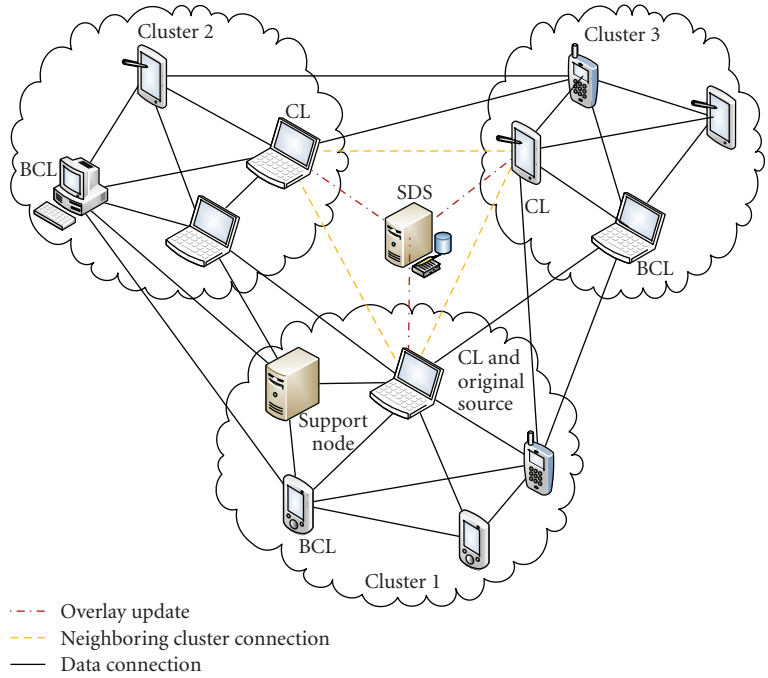

Figure 1: Overlay architecture.

creates multiple partial streams from an original RTP session. This implements multisource streaming in a way that each sender sends bursts of data from a different partial stream. Multisourcing will help to cope with the dynamics of mobile peers and distributes bandwidth usage in the system more flexibly and evenly. The original data stream source generates also RTP timestamps and sequence numbers for the RTP delivery. Timestamps and sequence numbers are delivered unchanged within the streaming service. This is done for allowing RTP packets from multiple partial streams being reassembled in the correct sequence order at each peer for local playback. The RTP time line is known system-wide, so the timestamps can be used to uniquely identify individual packets within a streaming service.

4. Overlay Network Architecture

The architecture of the overlay network with three clusters sharing a certain streaming service, such as a live stream channel or a VoD movie, is presented in Figure 1. It should be noted that for every different streaming service such an overlay network is maintained separately. The Service Discovery Server (SDS) is a central nonmobile server containing information about cluster hierarchy and the available streaming services in the system.

When a peer wants to join the P2P overlay network, a peer identifier (ID) is first requested from the SDS using an `RTSP OPTIONS` message with a `new_peer_id` tag in the `Overlay` header field. Because the peer does not have a peer ID yet, it must set the value to minus one in the `OPTIONS` message. A unique peer ID is then returned by the SDS using a `200 OK` message with a `New-Peer-Id` header field. The syntax of the `New-Peer-Id` header field in ABNF is given below:

```
New-Peer-Id = "New-Peer-Id:" SP id CRLF
id = 1*DIGIT
```

The cluster concept is implemented with the help of Cluster Leaders (CLs). There is one CL assigned to each cluster with the possibility for one or more Backup Cluster Leaders (BCLs). CLs are used to manage peers inside the cluster and to connect new arriving peers. Each ordinary peer must perform periodical keep alive messaging to inform its existence to the CL and all other peers from which it has received RTP packets. The latter is done to avoid unnecessary data transmission because RTP uses User Datagram Protocol (UDP) and the sending peer does not otherwise know that the receiving peer is still in the network. A new arriving peer can select a suitable cluster according to its best knowledge of locality using Round Trip Time (RTT) values between CLs and itself.

In addition to RTT measurements, location awareness could be also based on, for example, IP level hop count, geographic location or some combination of these three mentioned metrics. IP level hop count is not alone suitable for proximity metric, since with Virtual Private Networks (VPNs) or other tunneling techniques one hop might actually consist of a large number of hops and the distance could be quite long. Nor does small IP level hop count guarantee small delay, because it does not take connection speed into account. Geographic location is also little problematic in IP level point of view. Even if peers are geographically close to each other, the IP level routing path could circulate through distant router. Hence, only RTT values are used in our system for proximity checks.

CLs are nodes with suitable capabilities, such as a high throughput access network connection, enough memory and CPU power, and long-expected battery lifetime. One cluster should contain only a limited number of peers in order to sustain system scalability. The CL collects statistical data of the peers participating in a cluster. This statistical data contains information about service join time, reception buffer position, missing RTP packets, and upstream and downstream connections, and can be used to make the decision of the best peer from which to start downloading data. Statistical information can be used to, for example, filter out candidate source peers which already have many upstream connections or lots of missing RTP packets. Service join time can be used to estimate the behavior of the peer. If the peer has joined to the service very long time ago, it is most likely a stable peer which will provide data in the future also. On the other hand, without extra information about the expected battery lifetime with mobile devices, long service joining time can also mean short-expected battery lifetime.

The CL is a functional entity in the network and may also participate as an ordinary peer at the same time, by receiving and sending media data. Thus, the CL can be seen as a functional extension of an ordinary peer. The CL will inform the SDS currently at ten seconds intervals about changes in the cluster by sending an `OPTIONS` message with an `update` tag in the `Overlay` header field in order to maintain an up-to-date cluster list at the SDS. The updated cluster information will be expressed using an Extensible Markup Language (XML) [18]. To decrease the amount of data delivered in the network, all XML fragments are compressed using `deflate` compression mechanism from the zlib data compression library [19].

While joining the selected cluster, a peer receives an initial list of peers from which the actual media data can be acquired. Naturally, the corresponding CL inserts joined peers into its peer list, and if the amount of peers is very large, the CL can return only a subset of peers. Proximity testing in the peer selection is optional since the cluster selection procedure guarantees that peers are reasonably close to each other. Anyway, a peer which finally selects its sources for the stream, needs to test a certain amount of peers until suitable ones are found. The peer can later receive updates of the peer list while performing periodical keep alive messaging to the CL, which ensures that the peer list can be kept up-to-date during the streaming service.

The peer's contact information, that is, all information needed for contacting the peer, could include also a cluster ID, so that peers can prioritize connections within their own cluster. However, there should always be data connections between peers that are located in different clusters. This ensures that clusters do not become separate islands having only one incoming connection from other clusters, which would form a single point of failure that could cause problems later on when that peer leaves the streaming service.

4.1. Service Creation and Initial Cluster. The message exchange during service creation is presented in Figure 2. When a peer wants to create a service, an `ANNOUNCE` message will be sent to the SDS. A `Client-Port` header field indicates the port number to be used in the overlay communication. The service is described using the Session Description Protocol (SDP) [20]. Two new SDP attributes, `service-type` and `stream-info` are used to signal the service information. The `service-type` attribute defines the type for the service, and the `stream-info` attribute defines the identifier for the RTP session and parameters to be used in the RTP session partitioning explained in Section 7. The syntaxes for the `service-type` and `stream-info` attributes and the `Client-Port` header field in ABNF are given below:

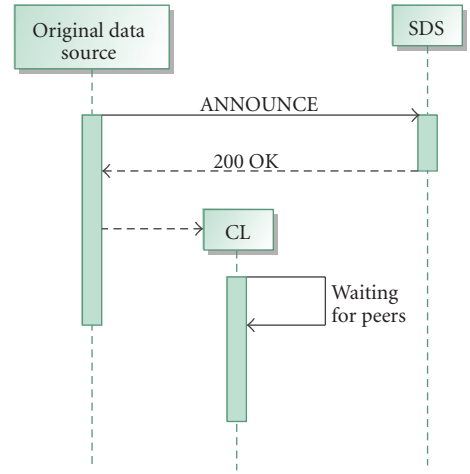

Figure 2: Creation of the service and the initial cluster.

```
service-type-line = "a=service-type:" type CRLF
type = "live" | "vod"

stream-info-line = "a=stream-info:" id";" piece-size";"
                   nb-of-partials ";" CRLF
id = "id=" 1*DIGIT
piece-size = "piece-size=" 1*DIGIT
nb-of-partials = "nb-of-partials=" 1*DIGIT

Client-Port = "Client-Port:" SP port CRLF
port = 1*DIGIT
```

As a response to the successful session creation, a `200 OK` message is sent by the SDS. The message contains the `Cluster-Id` and `Service-Id` header fields to describe the IDs for the initial cluster and the newly created service, respectively. A `301 Moved Permanently` message can also be sent if the SDS has been moved to another location. In a redirection case the `Location` header field must be present informing the new location of the SDS. Any other message type must be interpreted as a failed session creation. The syntaxes of the `Cluster-Id` and `Service-Id` header fields in ABNF are given below:

```
Cluster-Id = "Cluster-Id:" SP id CRLF
id = 1*DIGIT

Service-Id = "Service-Id:" SP id CRLF
id = 1*DIGIT
```

There are two possibilities for creating the initial cluster and selecting a CL for it: (a) the first peer joining the service

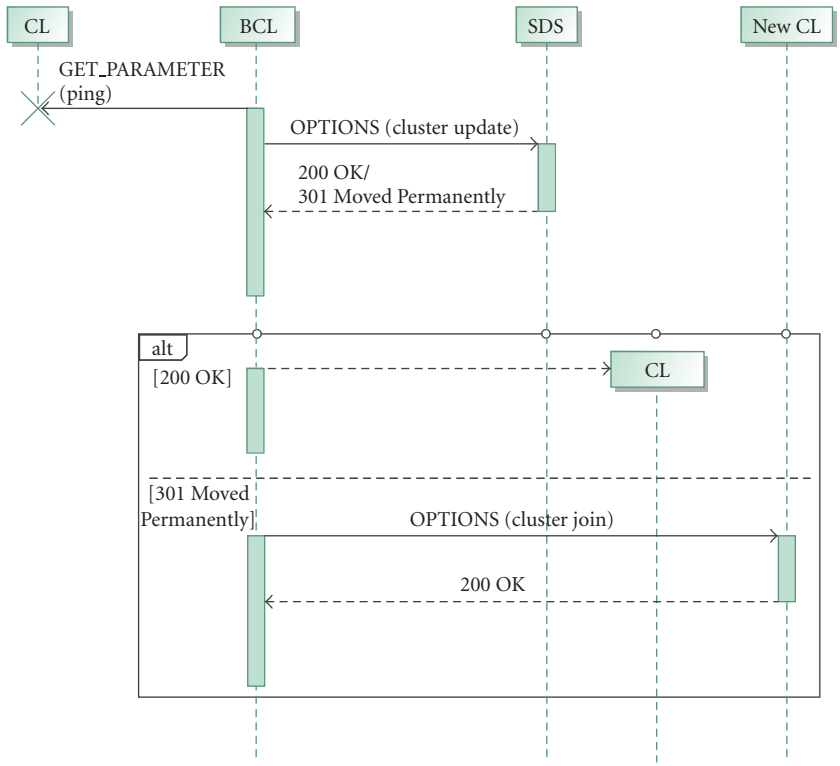

Figure 3: Uncontrolled CL departure.

will be assigned as a CL by the SDS, and (b) the original data source will be the first CL in the service. The latter possibility uses more resources from the original data source and therefore the original data source should be released from the CL responsibilities when possible. However, the latter possibility also guarantees that the initial cluster remains operational because the first joining peer might depart from the service quite quickly. Hence, the alternative (b) is used in our system.

When the service is successfully created, the original data source becomes CL for the initial cluster, which is illustrated by the dashed line without message type in Figure 2, and starts to wait for other peers to join the service. When new peers are joining the service, BCLs are assigned by the CL by using an `OPTIONS` message with a `backup` tag in the `Overlay` header field. If a peer accepts the BCL assignment, it sends a `200 OK` message, and if not, it will send a `403 Forbidden` message.

The service is updated and removed by using an `ANNOUNCE` message. If some part of the information have changed, the SDS updates the information in the database. To remove a service, the stop time in the SDP `t-line` should be set smaller than the prevailing system time, which means that the service has been stopped and the SDS can remove the service from the database. To a successful service update or removal the SDS will respond with a `200 OK` message, otherwise the SDS will return `400 Bad Request` or `404 Not Found` messages.

4.2. Cluster Leader Departure. When the CL leaves the network it needs to be replaced by one of the BCLs. If a cluster does not have an active CL, new peers cannot be accepted into the network. However, this does not affect the data streaming connections between existing peers because the streaming and overlay connections are independent. New peers cannot be discovered by normal peers during the cluster leader change, but this should not be an issue because peers should have knowledge about more peers than they are using.

The message exchange in the event of an uncontrolled CL departure is presented in Figure 3. When the BCL does not get a response to its periodical `GET_PARAMETER` message, it concludes that the CL has left from the cluster and contacts the SDS using an `OPTIONS` message with an `update` tag in the `Overlay` header field to replace the old CL. The source of the first received `OPTIONS` message will be assigned as a new CL, illustrated by the dashed line without message type in the figure, and the new arriving peers can normally start using the new CL. All other BCLs will receive a `301 Moved Permanently` message with the information about the new CL and will send an `OPTIONS` message with the `join_bcl` tag in the `Overlay` header field to the new CL and will continue in the BCL role. If the original CL has not left the cluster but has had connectivity issues, it is redirected to the new CL by the SDS. In this case the old CL becomes a BCL.

When a peer notices that the CL is not available, it tries to connect to the known BCLs. If the BCL has replaced the

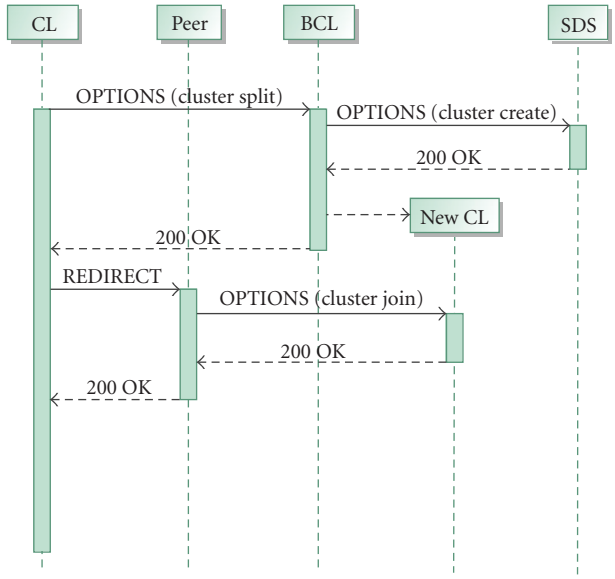

Figure 4: Successful cluster splitting procedure.

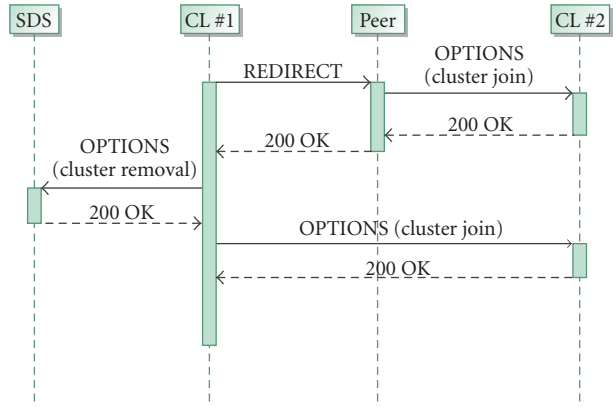

Figure 5: Successful cluster merging procedure.

old CL, it accepts the connection with a `200 OK` message, otherwise it sends a `301 Moved Permanently` message with the `Location` header field indicating the location of the last known CL. It should be noticed that the CL departure is not an atomic operation and takes some time, and therefore there can be short times when any one of the BCLs does not know the correct CL of the cluster. If none of the BCLs in the list respond, the peer sends a query to the SDS and asks for a new cluster which it can join.

4.3. Cluster Splitting and Merging. When the cluster grows too large to be handled by a single CL, the cluster should be split into two separate clusters. The existing CL assigns one of its BCLs to become a new CL for the new cluster, and redirects a number of existing peers to the new cluster. The message exchange in the successful cluster splitting procedure is presented in Figure 4.

Cluster splitting is performed by using an `OPTIONS` message with the `split` tag in the `Overlay` header field. After receiving this message, the BCL will inform the SDS that wants to become the CL of a new cluster by sending an `OPTIONS` message with a `create` tag in the `Overlay` header field. To a successful cluster creation, the SDS will respond with a `200 OK` message, which contains a `Cluster-Id` header field to describe the ID for the new cluster. Otherwise, the SDS will return a `400 Bad Request` message if the request message format is not valid, or a `404 Not Found` message if the service is not available anymore. After a successful cluster creation, the BCL will become the CL of the new cluster, and replies to the splitting CL by sending a `200 OK` message, otherwise it must send a `400 Bad Request` message to the splitting CL and wait for further instructions.

The splitting CL then sends a `REDIRECT` message, with the location of the new CL in the `Location` header field to those peers that should change the cluster. The redirected peers will then join the new cluster by sending an `OPTIONS` message to the new CL. After a successful cluster join, that is, the peer received a `200 OK` message from the new CL, the peer will send a `200 OK` message to the splitting CL. Otherwise, the peer will send a `400 Bad Request` message to inform that it is not possible to join to the new cluster.

The overlay connections between the CLs are created after a successful splitting by sending an `OPTIONS` message with a `join_neighbor` tag in the `Overlay` header field and a `200 OK` message. This connection is subsequently used to exchange cluster information expressed using XML fragments between neighboring clusters.

Merging of two clusters must be done when a cluster becomes too small. If the amount of peers is too small, a new joining peer will get a very small list of data sources which makes the functionality less reliable when one of these peers leaves the service. The message exchange in the successful cluster merging procedure is presented in Figure 5.

The merging is started by the CL, when the amount of peers in the cluster drops below some predefined threshold, by sending a `REDIRECT` message to all peers in the cluster. Peers will then join the new cluster, selected by the merging CL from its neighbor clusters, by sending an `OPTIONS` message to the new CL. After a successful cluster join, the peer will send a `200 OK` message to the merging CL. If the redirected peer does not receive any response from the new CL or it receives a `400 Bad Request` message, it must send a `400 Bad Request` message to the merging CL to inform that it is not possible to join to the new cluster and wait for further instructions.

After all peers in the cluster have confirmed the cluster change, the merging CL will remove the cluster by sending an `OPTIONS` message with a `remove` tag in the `Overlay` header field to the SDS. This message is optional, because the cluster is removed also automatically if a keepalive message has not been received during a certain interval. To a successful cluster removal, the SDS will respond with a `200 OK` message. Otherwise, the SDS will return a `400 Bad Request` message if the request message format is not valid, or a `404 Not Found` message if the cluster is not available anymore. The merging CL itself then must send a cluster join message, that is, an `OPTIONS` message with a `join_bcl` tag in

the `Overlay` header field, to a known neighbor and join as a BCL.

All overlay network connections are maintained by sending `GET_PARAMETER` and `200 OK` messages between peers as keep alive messages. Keep alive messages between neighboring CLs are exchanged at 20 seconds intervals and are used to exchange information about neighboring clusters. Keep alive messages between the CL and the BCL are used to deliver cluster information to the BCL and these messages are sent at 30 seconds intervals.

5. Partial RTP Streams

In order to have unique sending slots for each of the sources, a partial RTP stream concept is introduced in this paper. First, every RTP session, such as video, audio or subtitle streams of the entire multimedia session is split into smaller pieces along the time axis. Each of the pieces has a fixed duration T_P which is expressed in time. The start time is aligned with the start time of the RTP time base, that is, the start of the first piece is located at the origin of the RTP time line. One of the benefits of taking time as a unit is that all packets can remain intact at the RTP layer. Segmentation at the RTP packet level is not required for creating the partial streams. This significantly reduces the complexity of the implementation.

In the second step, RTP packets belonging to each of the RTP sessions are assigned to N partial streams according to (1), where i denotes the index of the partial stream ($0 \leq i < N$) and t_{RTP} denotes the RTP timestamp as carried in the RTP data packet. The algorithm allows assigning every RTP packet in the session to a partial RTP stream without having to maintain the state in the peer itself. Only by examining the RTP timestamp in combination with the constant parameters is sufficient to identify the partial stream the RTP packet is assigned to. Assuming a timeline of a single RTP session, this process is illustrated in Figure 6, where $T_C = N * T_P$ is used to denote the cycle time:

$$i = \text{floor}\left(\frac{t_{\text{RTP}}}{T_P}\right) \bmod N. \tag{1}$$

Note that the piece size T_P should be selected in such a way that it is large enough to contain at least one RTP packet on average. If it is chosen too small, not every piece will have data, which may in the extreme case lead to an empty partial stream. On the other hand, larger cycle times lead to longer startup times, since a complete cycle needs to be buffered before seamless playback can be guaranteed. In one particular type of partitioning, every piece would start with an intracoded picture. This would facilitate independent decoding of partial streams in the presence of packet loss due to the fact that a partial stream is not being received. This could for instance easily be achieved by aligning the pieces with group-of-picture (GOP) boundaries. Enhanced robustness will also be achieved by assigning key (RTP) packets to multiple partials. Key packets could for instance be Instantaneous Decoding Refresh (IDR) picture data or other data that would help error concealment. Duplicate RTP packets would simply be removed upon reception and would therefore not affect the basic algorithm or its complexity.

The number of partial streams, N, can vary per RTP session. For instance it may not be very useful to partition an audio stream into lower bit rate partial streams if the bit rate of the entire RTP audio session is already in the order of magnitude of a single partial RTP video stream. This also has the additional advantage that the audio is received either in its entirety or not at all, thereby reducing annoying audible artifacts in the case of partial stream loss.

The number of partial streams does not necessarily need to be constant throughout the P2P network within a particular streaming service. As a matter of fact it is possible to vary N at every forwarding peer in the network. However, choosing the same N throughout the network simplifies the design of the partitioning functionality.

6. Media Delivery Mechanism

All peers in the streaming network are forming a non-hierarchical mesh structure. Peers are connected to several other peers and are receiving data from and sending data to multiple other peers. An example mesh layout can be seen in Figure 1.

A peer may request the delivery of one or more partial streams from another peer. A partial stream is the smallest granularity for media streaming, that is, a peer may not stream a fraction of a partial stream. The number of partial streams can be tuned to achieve the target bit rate of a partial stream. Each peer in the network should have enough uplink bandwidth to be able to stream at least a single partial stream.

In this kind of streaming network, where most of the peers are from the same cluster, some kind of intelligence to avoid loops is needed. Such loops occur when a sender starts receiving its own data via a number of intermediate peers in the mesh network. To avoid loops, an algorithm based on streaming path in the form of list of ancestors is used. The path for the data stream in the application level containing peer IDs which have forwarded the stream is delivered using the contributing source (CSRC) list in the RTP packets. This list is then used to avoid accepting connections from peers who are already in the list and for dropping connections if a peer notices that it is in the list.

Figure 7 illustrates media delivery among four peers. Arrows between each peer denote an active RTP session, and the direction defines the data flow direction. A sourcing peer can send multiple partial streams to a particular receiving peer. This allows for a smaller granularity of rate adaptation between two individual peers in the network. These multiple partial streams could either be streamed in a single RTP session or separate RTP sessions. Peers in the figure are numbered and colored. The smaller the number is, the earlier the peer has joined the network; in this case peer number one is the original source. Different colors in the peers buffers show the origin of the received data. For simplicity, the value four is used for the number of partials for each peer.

In order to receive a complete stream, a peer must receive all partial streams; in this example, partials 0, 1, 2, and 3.

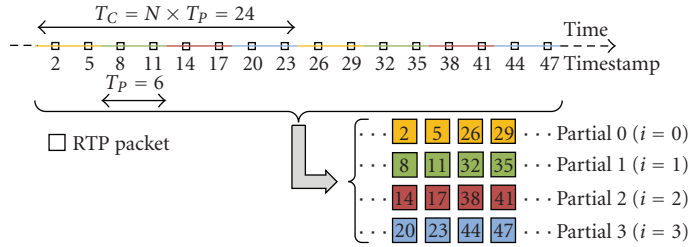

Figure 6: RTP stream partitioning.

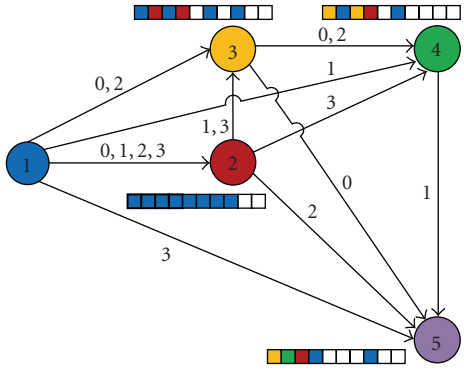

Figure 7: Partial RTP stream delivery.

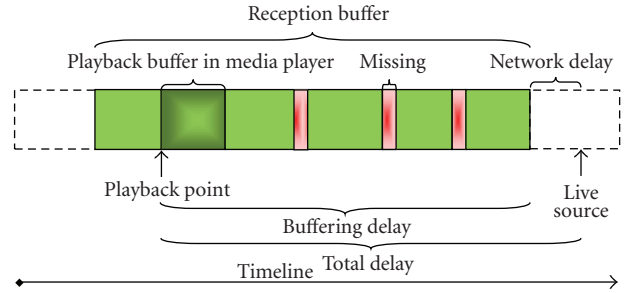

Figure 8: Data buffering.

Peer number two is receiving all partials from the original data source, that is, peer number one, and is forwarding partials to peers number three, four, and five. Peer number four is also receiving partials from peer number three, and peer number five from peers number three and four. All incoming packets for all of the partial streams that constitute a particular RTP session are added to a separate buffer pool. This buffer pool maintains the information about the destination peer to which the incoming packets need to be forwarded. Every incoming packet is examined and assigned to one of the outgoing queues. In case the peer is playing back the received streams locally, the local player is considered to be a destination as well.

Packets destined for a particular receiving peer are transferred from the buffer to the outgoing RTP queue as soon as they have been received. In case of local playback, that is, the receiving peer is decoding and rendering the received RTP session, the packets are also forwarded internally to the media player. Buffering is needed to recover from peer departures and consequential missing data. The peer can ask a replacement peer to send the missing data from its buffer. There is also a possibility that a peer does not have certain part of the buffer available because of bad network conditions, for example, insufficient bandwidth. These parts might be later downloaded if needed. Because of the missing data, the playback is interrupted during this period and it depends on the used codec how missing data will affect on Quality of Experience (QoE). Data will most probably not come in order or in time from all sources as the delay could vary very much between different sources, so the buffer is also needed to collect all the data from different partial streams and arrange the data in order before passing it to the local media player.

When all partial streams are received almost at the same time, the buffering delay can be reduced and the data can be passed earlier to the media player. However, the situation may change during time, and a constant buffer delay is currently used in the implementation. Data buffering is presented in Figure 8. The playback buffer is the buffer located in the media player. When peers consume media, the reception buffer is simultaneously shared with other peers. The total delay from content generation to receiver playback is the sum of the network delay, buffer delay, and media player playback delay.

In addition to the reception buffer, data storage via caching should be used in a VoD streaming service. When using caches, the VoD data can be distributed away from the original data source. This helps relieving the network load from the original data source, as new peers joining the network are able to download VoD content from multiple sources instead of relying on the original data source. Caching could be implemented in many ways, in the simplest model all peers store all data they have consumed. If a peer does not have enough storage capacity, the data belonging to a specific partial stream could be cached instead. Alternatively, a peer can limit the amount of cached data by applying a sliding time window. In the former case, an algorithm based on peer IDs could be used to determine which partial stream should be stored. This kind of partial caching requires that the amount of peers in a streaming service is reasonably high. To ensure data availability from multiple peers in every situation, support nodes, which

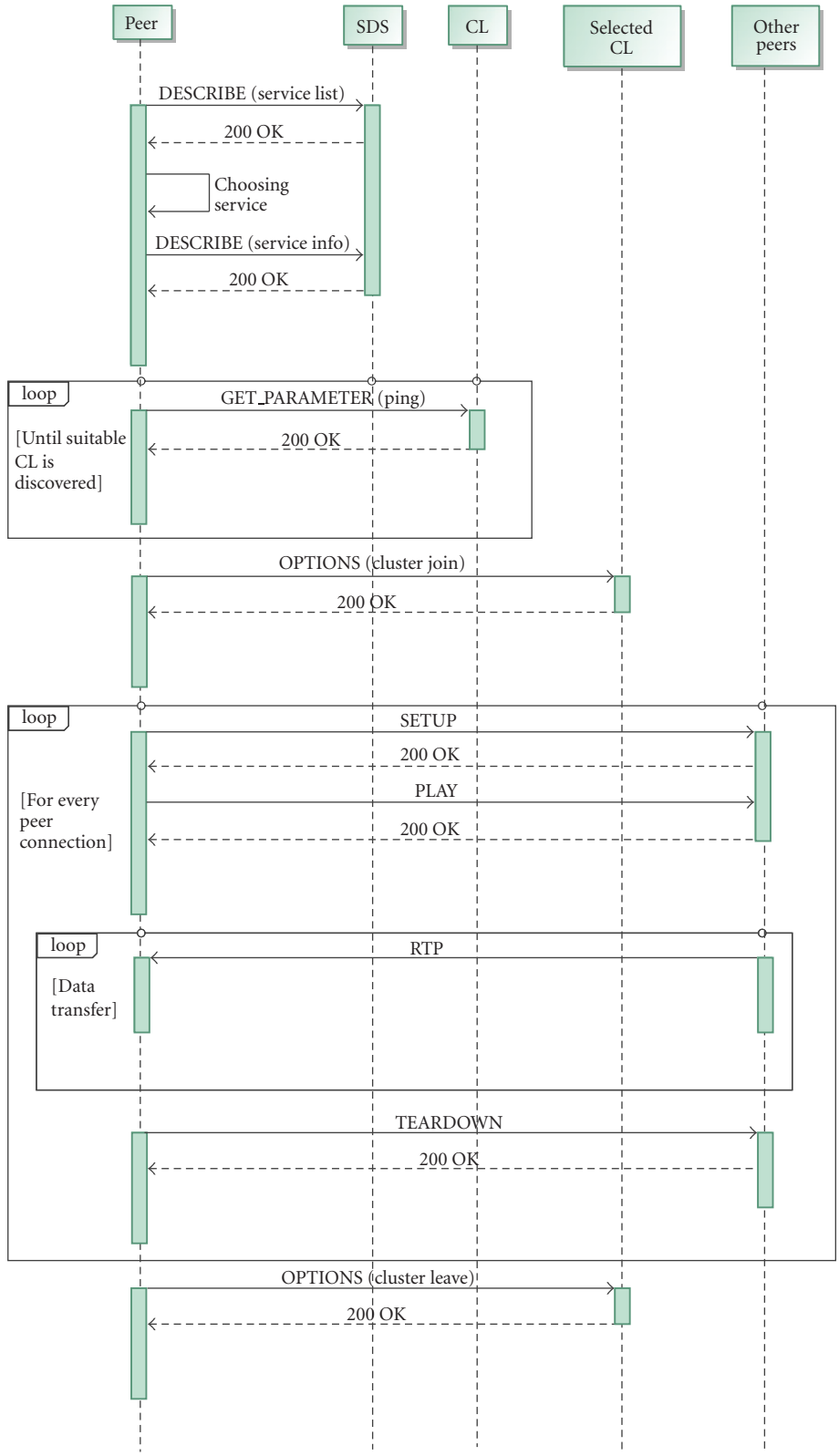

Figure 9: Peer in a service.

act as stream relay nodes, could be used to store all the data. Such relay nodes simply store the data and pass it to other network participants. It should be noted that a support node also consumes upload bandwidth from other peers, but with a high throughput network connection it provides more bandwidth to others than it consumes. The currently existing simple VoD implementation uses the simplest caching model, which is of course not the most optimal one, but it enabled a fast proof-of-concept implementation.

7. Peer Operation in a Service

Figure 9 shows the message exchange in case a peer is participating in a particular service. For getting a list of all available services a `DESCRIBE` message is sent to the SDS. The service list information obtained in the `200 OK` message contains only general information of the services to decrease the message size and the information is expressed as XML fragments. If a user wishes to search services with a wildcard string, a search element could be used to deliver the wildcard string to the SDS. If there are not available services, the SDS will respond with a `404 Not Found` message.

To be able to join to a particular service, more detailed service information must be retrieved from the SDS using a `DESCRIBE` message with an RTSP URL specifying the service. A `200 OK` message contains only a partial list of the available clusters, in case a large number of clusters has been created. The response uses multipart MIME [21], because it must deliver both the SDP of the service and the initial cluster list in XML format. If the service is not available anymore, a `204 No Content` message will be sent by the SDS.

After obtaining the CL list from the SDS, the peer makes contact with several CLs until a suitable CL is discovered. For this purpose, the peer sends a `GET_PARAMETER` message to the CL and starts the RTT counter. The peer stops the RTT counter when it receives a `200 OK` message and compares the RTT value to some predefined upper limit. The first CL with an RTT value below the limit is then selected as the CL for the service.

The peer joins the selected cluster by sending an `OPTIONS` message with a `join_peer` tag in the `Overlay` header field to the CL of the cluster. In the `200 OK` message, an initial peer list is received in XML format if some peers have already joined the cluster. Otherwise, a `Cluster-Id` header field is used to describe the ID for the cluster. The initial peer list is a random subset of the total peer set, if there are lots of peers in the cluster.

The peer tries to request data from other peers in the list order using a `SETUP` message. This message handles the configuration of the UDP port numbers for the RTP reception using the `Transport` header field. If there are fewer peers than the target number of partial streams, the peer continues requesting again from the beginning of the list, so that it will receive more than one partial stream from a single peer. If a certain peer is not responding, it will be removed from the internal *known peer list*, so that the peer does not try to reconnect again. The peer which is receiving the requested stream, that is, audio or video stream, will respond with a `200 OK` message to indicate that it might be possible to get the stream data from the peer.

Setting up of the partial RTP streams is done by sending a `PLAY` message to the peer which is receiving the requested stream. Splitting of the original RTP session into partial streams is explained in Section 5, and these `partial stream` parameters are signaled using a `Partial-Stream` header field. The format of a `Partial-Stream` header field in ABNF is given below:

```
Partial-Stream = "Partial-Stream:" SP partial-stream-
                 info
                 CRLF

partial-stream-info = id ";" piece-size ";"
                      nb-of-partials ";"
id = "id=" 1*DIGIT
piece-size = "piece-size=" 1*DIGIT
nb-of-partials = "nb-of-partials=" 1*DIGIT
```

If the peer is able to send the requested partials, it will respond with a `200 OK` message. If the peer noticed a loop, it will response with a `400 Bad Request` message, and if the peer cannot send the requested partials, it will respond with a `404 Not Found` message. After the `200 OK` message, data delivery from the requested peer using RTP is started. If the interval between two consecutive RTP packets is more than the predefined maximum allowed delay, the receiving peer should conclude that the sending peer is not capable of delivering the data in time and it should change the sender for the partial in question.

A peer can depart from the network in two ways. In a controlled departure, the peer informs its neighbors and the CL that it is leaving the network. The peer sends an `OPTIONS` message with a `leave`, tag in the `Overlay` header field to the CL, and a `TEARDOWN` message to all of its data delivery neighbors. Neighbors, which were sending data know that they can terminate the RTP session. Also peers that were receiving data know that they will not be able to receive more data from that peer, and can search for a replacement. The `TEARDOWN` message will also be sent when a peer notices that there is a loop in the data delivery for some partial stream.

An uncontrolled peer departure is noticed both by the CL and a peer which sends data to the departed peer after connection keep alive messages, that is, `GET_PARAMETER` messages, have not been received within some time interval X. Currently, keep alive messages are sent at 30 seconds intervals towards the CL and at 15 seconds intervals towards the peer which sends the data. If the sending peer does not receive keep alive message within 30 seconds interval it concludes that the receiving peer has departed and terminates the RTP session. Similarly, if the CL does not receive any message from the peer within 45 seconds interval it concludes that the receiving peer has departed and removes the peer from the cluster because of inactivity. A peer which is receiving data from the departed peer will notice uncontrolled departure after it has not received any RTP packets since Y seconds. The value Y should be defined so that it is possible to get data from a replacement peer within the reception buffer duration in order to avoid interruption. This is basically the same situation when the sending peer

is not capable of delivering the data in time and currently Y is calculated according to (2). This value consists of time between two pieces belonging to the same partial RTP stream, the normal network delay T_N, that can be calculated from RTSP request-response pairs, and a small extra time T_E given to peers to patch packets that might still be forwarded in the network:

$$Y = T_P * (N - 1) + T_N + T_E. \quad (2)$$

To request data from a replacement peer from a certain starting point, a `Packet-Range` header field can be included into a `PLAY` message to signal the play-after value using RTP sequence numbers. The `Packet-Range` header field can also be used to signal the current playback position when the peers are seeking a new playback position in a VoD service. The format of a `Packet-Range` header field in ABNF is given below. The two different use cases can be distinguished by the minus sign used in the former case:

```
Packet-Range = "Packet-Range:" SP range-specifier CRLF
range-specifier = 1*DIGIT ["-"]
```

In the VoD service, two other additional header fields are required for the seeking operation. The desired seek time in milliseconds will be signaled using a `Seek` header field. A `Fast-Send` header field will be used to inform the sender to send a specified amount of data (in milliseconds) as fast as possible to be able to fill up the reception buffer and start playback with as small delay as possible. The formats of the `Seek` and a `Fast-Send` header fields in ABNF are given below:

```
Seek = "Seek:" SP seek-time CRLF
seek-time = 1*DIGIT

Fast-Send = "Fast-Send:" SP fast-send-time CRLF
fast-send-time = 1*DIGIT
```

8. Implementation

The architecture of the real-time P2P streaming (RTP2P) application is presented in Figure 10. The RTP2P application is implemented using C++ and it consist of ten different software components. The principle in the figure is that the higher layer uses all components that are immediately below it, so, for example, the Graphical User Interface (GUI) uses Service, Common, and Media Player components.

The GUI is implemented using the gtkmm framework [22] for the Linux desktop environment and the maemomm framework [23] for the Nokia N800 Internet Tablet. Currently three different media players, VLC [24], MPlayer [25], and GStreamer [26], are needed in the application. This is necessary because any single player cannot offer all the features required for our application. VLC is used to stream RTP packets locally to the RTP2P application in the case of the original data source. The original data source only listens to the local socket, and receives RTP packets generated by the VLC and can forward those further using the multisource streaming concept explained in Sections 5 and 6. MPlayer is used for the media playback on the client applications. It is also possible to use VLC for the media playback, but with MPlayer it is possible to achieve a better synchronization with multiple elementary streams by using RTCP sender reports. GStreamer is used to create an RTP stream from the camera of the N800 device. The needed RTP operations are provided by the GNU ccRTP library [27]. In addition, the Boost libraries [28] are utilized for threading time and file system operations.

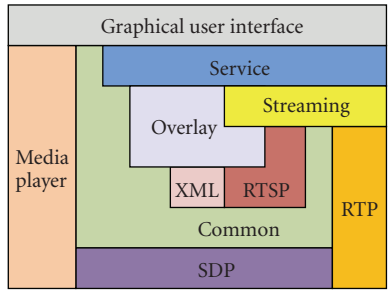

Figure 10: Architecture of the RTP2P streaming application

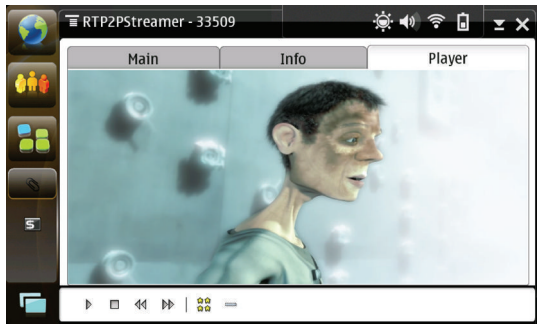

Figure 11: Graphical user interface.

Other proprietary software components, that is, Service, Overlay, Streaming, XML, RTSP, Common, and SDP, form the basis for the peer operation. The SDP component is based on the GNU oSIP library [29] and is used to parse the streaming service description expressed in the SDP format. The Common component contains definitions and functionalities which are widely used by other components including, for example, threading and socket operations. The XML component is based on the Expat XML parser [30] and it is utilized for parsing cluster and service information expressed in the XML format. The RTSP component contains RTSP message creation and parsing and also the base functionality for RTSP operations, which are enhanced and utilized by Streaming and Overlay components. The Streaming component includes sender and receiver functionalities to handle P2P RTSP communications and RTP reception and sending operations for the streaming service. The functionality of the cluster leader and all RTSP overlay communication are included in the Overlay component. The Service component contains the functionality needed to join, create, and manage streaming services.

Figure 11 presents the GUI in a Nokia N800 device. The GUI consist of three parts: (a) the main application view

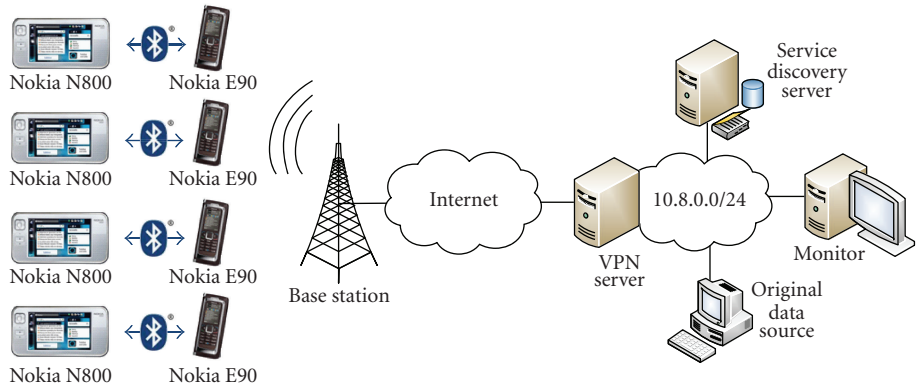

Figure 12: Test setup for evaluating the operation of the system in the mobile environment.

with Main, Info and Player tabs, (b) Create, Connect, List and Preferences dialogs launched from the drop down menu, and (c) Toolbar buttons at the bottom to start and stop the selected service at the receiving end, and the service in the original data source. The List dialog is used to obtain the service list from the SDS which is then shown in the Main tab. Information about the sourced service is also shown in the Main tab. The Info tab is used to show detailed service information about the sourced service and the service that is currently received by the peer. The Player tab shows the sourced stream in the original data source and the currently received stream by the peer. The Connect dialog is used to enter an IP address or a domain name for connecting to the SDS. The Create dialog is used for creating new services. Service name, type and description are given by the user, as well as the file to be sourced in case of stored content. The content can also be captured directly from the camera. The Preferences dialog can be used to adjust some attributes, for example, port numbers for the network traffic and the level of debug messaging.

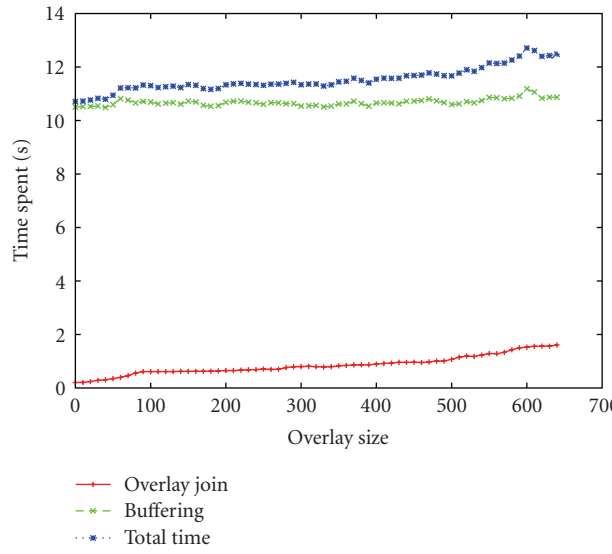

Figure 13: Overlay joining time and an initial buffering time as a function of overlay size; maximum cluster size 30 peers.

9. Performance Evaluation

The test setup for evaluating the operation of the system in the mobile environment is presented in Figure 12. Four Nokia N800 Internet Tablets with HSDPA network connection provided by the Nokia E90 were used together with a PC acting as an SDS. In some test cases, also another PC was acting as an original data source. During the tests a special application was used to monitor streaming connections between peers. As most normal consumer connections, also mobile connections are suffering from Network Address Translation (NAT) based connection limitations because Internet Service Providers (ISPs) do not want that users use the relatively small mobile bandwidth for hosting services or using P2P technologies. To avoid connection limitations, a workaround with VPN solution was utilized. Every entity with a restricted network connection first creates a connection to the VPN server and gets the desired free connectivity through the VPN tunnel. A more permanent solution could be implemented using NAT traversal to overcome these restrictions.

Tests have shown that the system performs well over mobile connections with ten seconds initial buffering time. The buffer size in our application is also ten seconds due to the RTP usage and is relatively small compared to the buffer sizes of existing solutions reported in [4].

Because the amount of available mobile devices was limited, a laboratory network environment with 17 desktop PCs (Intel Core2Duo E6550, 4 GB DDR2, running CentOS Linux, kernel 2.6.18 PAE) has been utilized to test the system functionality with a larger amount of peers. 16 hosts were used to run 40 peers in each host together with one host acting both as an SDS and as an original data source. The connectivity between all devices was provided by a 1 Gbps switch. The length for one live streaming service was roughly one hour and all forthcoming figures present average values from five different live streaming services. The maximum cluster size was set either to 30 or to 70 peers, and peers were started in 40 cycles with a 5 s starting interval: first one peer was started at each host, then a second one and so on.

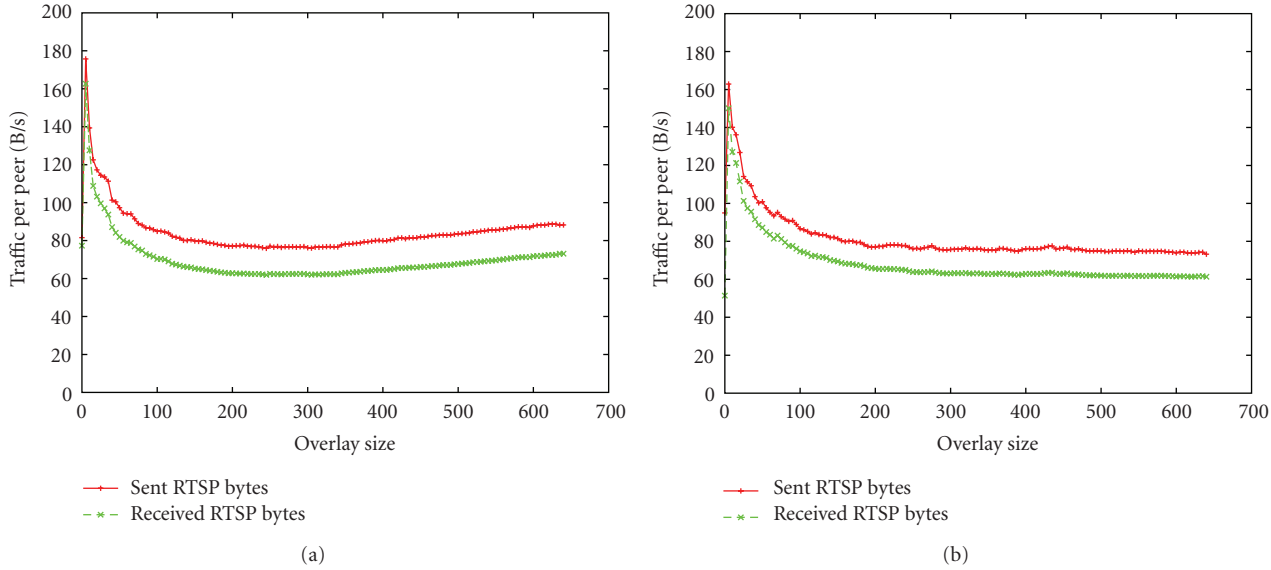

Figure 14: The amount of sent and received RTSP data, (a) maximum cluster size 30 peers, and (b) maximum cluster size 70 peers.

9.1. Steady Network. In this test scenario all peers stayed steadily in the service from the joining time to the end of the service. The streaming service joining time, that is, the overlay joining time plus the initial buffering time, as a function of the overlay size is presented in Figure 13. From the figure we can see that the initial buffering time remains almost constant regardless of the number of peers in the network. The increase in overlay joining time could be minimized by improving the current cluster selection algorithm.

The amount of sent and received RTSP data in bytes per peer as a function of overlay size is shown in Figure 14. The combined bit rate of the original RTP sessions is about 112 kbps, encoded using FFmpeg's [31] H.263+ video, and AAC audio codecs. Hence, the RTSP signalling overhead is quite minimal compared to the actual media data. The reason for the larger amount of RSTP data sent is caused by the relatively large cluster status information messages transmitted to the SDS by the CLs. The main difference between Figure 14(a) and Figure 14(b) is the slight increase in the RTSP signalling data after 300 peers with the maximum cluster size of 30 peers. This is caused by the larger (and increasing) amount of clusters and cluster status information messages. Similar effect might take place also with the maximum cluster size of 70 peers when the amount of peers is increased above the currently used maximum value.

9.2. Network with Leaving and Rejoining Peers. To simulate even a more realistic situation and the churning caused by the real mobile nodes, a timer functionality that is able to randomly shut down and restart nodes was used. After a peer had joined to the service, it stayed randomly from 30 s to 10 min in the service and then left the service and joined back after 10 s. This allowed us to test the network in more realistic situations where peers are leaving and data connections are failing.

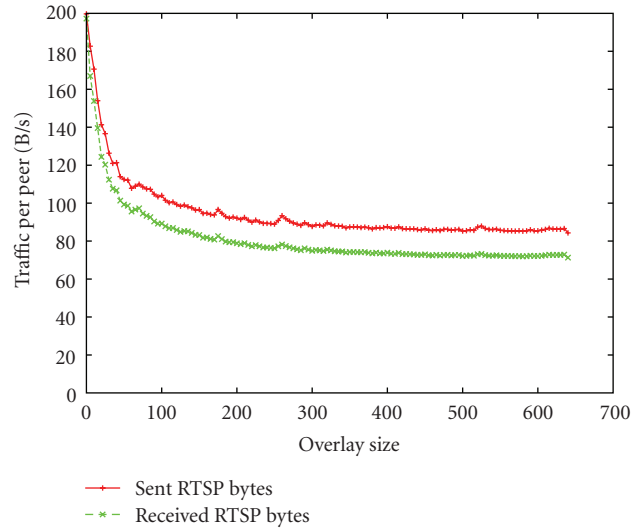

Figure 15: The amount of sent and received RTSP data with leaving and rejoining peers; maximum cluster size 70 peers

The amount of sent and received RTSP data in bytes per peer as a function of the overlay size is presented in Figure 15. If we compare these values to the corresponding values in the steady state scenario, we can see that the values follow the same trend, but the replacement peer searching and peer rejoining messages cause small overall increase to the signalling data.

10. Future Developments

In the current overlay implementation, a cluster change is controlled by the CL. Peer-controlled cluster change, where a peer changes the cluster after acquiring knowledge about a new cluster which would serve its data requirements better,

would make the system more scalable and will increase the overall performance.

Clusters are currently loosely connected together between the neighboring CLs to share some peer information. A clearer cluster group structure with a Cluster Group Leader (CGL) is worth studying. The CGL could collect the information about all clusters within the cluster group and send cluster update messages to the SDS, instead of separate update messages from individual CLs. This organization into an n-level hierarchical structure could increase network scalability and reduce the overlay joining time since the cluster search time would be reduced from $O(n)$ to $O(\log(n))$. As a drawback, the complexity of the system and the overlay maintenance will be highly increased.

More advanced implementation level support for VoD streaming such as better caching mechanism and support for other Video Cassette Recording (VCR) functionalities like fast-forward and rewind, in addition to the currently existing seek functionality will definitely pose different requirements compared to the live streaming case. Mechanisms for handling packet losses is an important research area in peer-to-peer streaming. Different error robustness techniques, such as simple retransmission, Forward Error Correction (FEC) and network coding, need to be studied to find out the benefits and drawbacks of those techniques, when used in addition to the current mechanism which is based on peer replacement before the reception buffer underflows.

One interesting research area is the usage of Multiple Description Coding (MDC) [32] or Scalable Video Coding (SVC) [33] in the real-time P2P streaming as is proposed also in [10]. A single stream is divided into several descriptions and each of the descriptions is then forwarded separately to the network. With this approach, the current partial stream could be replaced by one description without affecting the clustered overlay network architecture. However, our partial stream concept has much lower complexity (than MDC or SVC), which has enabled a fast proof-of-concept implementation. Our design allows an easy replacement of the partial stream concept with MDC or SVC as soon as their implementations become publicly available.

11. Conclusions

The effective real-time P2P streaming system for the mobile environment presented in this paper is an alternative solution to traditional client-server-based streaming applications. A scalable overlay network which groups peers into clusters according to their proximity is created and maintained using extended RTSP messages by the cluster leaders with the help of a service discovery server. Furthermore, the actual media delivery is implemented using a partial RTP stream concept. RTP sessions are split into a number of partial streams in such a way that it allows reassembling the original media session in real-time at the receiving end.

The first laboratory tests together with the tests in the mobile environment have shown that the current implementation performs well and offers very low initial buffering times. More advanced laboratory tests with different latencies and throughputs between peers are still needed to highlight system bottlenecks and usability issues.

Acknowledgments

The authors would like to thank Joep van Gassel, Alex Jantunen and Marko Saukko for their valuable work as part of the development team. This work was partially supported by TEKES as part of the Future Internet program of Finnish Strategic Centre for Science, Technology and Innovation in the field of ICT (TIVIT).

References

[1] "YouTube—Broadcast Yourself," May 2009, http://www.youtube.com/.

[2] "Octoshape," May 2009, http://www.octoshape.com/.

[3] "SopCast," May 2009, http://www.sopcast.org/.

[4] J. Peltotalo, J. Harju, A. Jantunen, et al., "Peer-to-peer streaming technology survey," in *Proceedings of the 7th International Conference on Networking (ICN '08)*, pp. 342–350, April 2008.

[5] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications," Internet Engineering Task Force, RFC 3550, July 2003, http://www.rfc-editor.org/rfc/rfc3550.txt.

[6] B. Cohen, "Incentives build robustness in BitTorrent," in *Proceedings of the Workshop on Economics of Peer-to-Peer Systems (P2PECON '03)*, pp. 116–121, June 2003.

[7] P. Shah and J.-F. Paris, "Peer-to-peer multimedia streaming using BitTorrent," in *Proceedings of the 26th IEEE International Performance, Computing, and Communications Conference (IPCC '07)*, pp. 340–347, April 2007.

[8] X. Jiang, Y. Dong, D. Xu, and B. Bhargava, "GnuStream: a P2P media streaming system prototype," in *Proceedings of the International Conference on Multimedia and Expo (ICME '03)*, pp. 325–328, July 2003.

[9] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum, "CoolStreaming/DONet: a data-driven overlay network for peer-to-peer live media streaming," in *Proceeding of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '05)*, vol. 3, pp. 2102–2111, March 2005.

[10] N. Magharei and R. Rejaie, "PRIME: peer-to-peer receiver-drIven MEsh-based streaming," *Proceedings of the 26th IEEE International Conference on Computer Communications (INFOCOM '07)*, pp. 1415–1423, May 2007.

[11] D. A. Tran, K. A. Hua, and T. Do, "ZIGZAG: an efficient peer-to-peer scheme for media streaming," in *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '03)*, vol. 2, pp. 1283–1292, March 2003.

[12] J. Liang and K. Nahrstedt, "DagStream: locality aware and failure resilient peer-to-peer streaming," in *Proceedings of the 13th Annual Multimedia Computing and Networking Conference (MMCN '06)*, pp. 224–238, January 2006.

[13] J. Yu and M. Li, "CBT: a proximity-aware peer clustering system in large-scale BitTorrent-like peer-to-peer networks," *Computer Communications*, vol. 31, no. 3, pp. 591–602, 2008.

[14] J. Peltotalo, J. Harju, M. Saukko, et al., "A real-time peer-to-peer streaming system for mobile networking environment," in *Proceedings of the INFOCOM and Workshop on Mobile Video Delivery (MoVID '09)*, April 2009.

[15] H. Schulzrinne, A. Rao, and R. Lanphier, "Real Time Streaming Protocol (RTSP)," Internet Engineering Task Force, RFC 2326, April 1998, http://www.rfc-editor.org/rfc/rfc2326.txt.
[16] D. Crocker and P. Overell, "Augmented BNF for Syntax Specifications: ABNF," Internet Engineering Task Force, RFC 2234, November 1997, http://www.rfc-editor.org/rfc/rfc2234.txt.
[17] T. Berners-Lee, R. Fielding, and L. Masinter, ""Uniform Resource Identifier (URI): Generic Syntax," Internet Engineering Task Force, RFC 3986, January 2005, http://www.rfc-editor.org/rfc/rfc3986.txt.
[18] W3C, *Extensible Markup Language (XML) 1.0*, World Wide Web Consortium (W3C), 4th edition, 2006.
[19] "zlib," May 2009, http://zlib.net/.
[20] M. Handley and V. Jacobson, "SDP: Session Description Protocol," Internet Engineering Task Force, RFC 2327, April 1998, http://www.rfc-editor.org/rfc/rfc2327.txt.
[21] K. Moore, "MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text," Internet Engineering Task Force, RFC 1522, Sepember 1993, http://www.rfc-editor.org/rfc/rfc1522.txt.
[22] "gtkmm—C++ Interfaces for GTK+ and GNOME," May 2009, http://www.gtkmm.org/.
[23] "maemomm—C++ bindings for the Maemo API," May 2009, http://maemomm.garage.maemo.org/docs/index.html.
[24] "VLC Media Player," May 2009, http://www.videolan.org/vlc/.
[25] "MPlayer—The Movie Player," May 2009, http://www.mplayerhq.hu/.
[26] "GStreamer: open source multimedia framework," May 2009, http://www.gstreamer.net/.
[27] "GNU ccRTP—GNU Telephony," May 2009, http://www.gnu.org/software/ccrtp/.
[28] "Boost C++ Libraries," May 2009, http://www.boost.org/.
[29] "The GNU oSIP Library," May 2009, http://www.gnu.org/software/osip/osip.html.
[30] "The Expat XML Parser," May 2009, http://expat.sourceforge.net/.
[31] "FFmpeg," May 2009, http://www.ffmpeg.org/.
[32] V. K. Goyal, "Multiple description coding: compression meets the network," *IEEE Signal Processing Magazine*, vol. 18, no. 5, pp. 74–93, 2001.
[33] H. Schwarz, D. Marpe, and T. Wiegand, "Overview of the scalable video coding extension of the H.264/AVC standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 17, no. 9, pp. 1103–1120, 2007.