Design of Distributed Collaborative Application through Service Aggregation

Andrew Roczniak, Jamil Melhem, Pierre Lévy and Abdulmotaleb El Saddik
Multimedia Communications Research Laboratory
Collective Intelligence Laboratory
University of Ottawa, Ottawa, Canada
Email: \{roczniak, jmelhem, abed\}@mcrlab.uottawa.ca, plevy@uottawa.ca

Abstract

The Service Oriented Architecture (SOA) is used to support loosely-coupled integration of existing applications. We are investigating the possibility of creating entirely new applications based on SOA. We present a case study for a collaborative authoring application targeting groups of around five users collaborating over the Internet. We highlight the basic requirements of the application and show how these can be fulfilled by utilizing certain services accessed through standard HTTP, Jabber and JXTA set of protocols, and off-the-shelf techniques. By measuring the performance of the application in a heterogeneous environment and by providing details of an alternate service fulfilling the application’s requirement, we show that SOA-based collaborative applications can be quickly designed and deployed.

1 Introduction

Developing a collaborative authoring application presents numerous challenges, one of them being that users’ requirements may change faster than the development and distribution cycle of any application designed to meet their needs. Inspired by service-oriented architectures (SOA), we investigate the possibility of empowering users to quickly assemble their own application from a set of standard services. A service is some functionality that is packaged and offered in a specific way. Services are self contained and operate as black boxes; all necessary components to accomplish a specific functionality are encapsulated within the service and consumers are not concerned with the underlying implementation. Those characteristics should make a service independent of consumer context, allowing reuse in contexts not known at design time. Services expose their functionality through interfaces. The requirement on those interfaces is that they be invokable in a standardized way. The service itself may be within the same application or in a completely different system on the Internet but the interconnect scheme or protocol used to allow the invocation should however be open and standardized. This in turn requires open and well-understood way of publishing, finding and binding to services. The infrastructure components required to make the connection can be diverse since services may be implemented on a single machine or distributed across a set of computers on a network [10]. Services can be composed into other services - based on quantifiable quality of service, for example - and offered transparently to the consumer, as one service. Traditionally, SOA implementation relies on the Web Services (WS) family of protocols to support interface definition and binding (WSDL), message exchange (SOAP), service discovery (UDDI) and composition techniques as presented in [5]. We have elected to use the Jabber and JXTA set of protocols instead of the WS protocols due to their relative simplicity and their closer focus on the functionality we are developing. To verify that the usage of Jabber and JXTA protocols is consistent with the general understanding of SOA, we are relying on compliance to the Reference Model for Service-Oriented Architecture as specified by OASIS [1].

At its basis, a collaborative authoring tool has at the minimum three main requirements: bulk transfer (to share files), group notification (to notify other users about one’s actions) and document consistency (to ensure that the shared objects are consistent between users. The problem is thus to select services that fulfill those requirements while being consistent with SOA philosophy. Responsiveness, the time necessary for the system to respond to users’ actions, and fidelity, the degree to which the end result is similar for all users, form the basis of our evaluation criteria.

The jabber.org server (Jabber server) provides various services accessed through the Jabber set of protocols. There are three services that we will be using: a Jabber client can send short XML-based messages to any other connected client, Jabber servers usually provide chat rooms supporting multiple participants where any message issued to the
chat room will be forwarded to all participants that joined it, and finally, since each Jabber client needs to establish a connection with a server, this presence information can be made available (depending on security and privacy policies in effect) to other clients. Note that we are not using certain functionalities such as chat room playback and potential server-side components as they are defined and controlled by the Jabber server administrator and are thus not standardized. Similarly, we use the JXTA set of peer-to-peer protocols to publish service advertisements at a Rendez-Vous (RV) peer, and to establish a direct connection between any two participants. Services are peer-based (RV acts as a broker), and we are mainly interested in the ability to exchange large amount of data between any two peers.

The group notification requirement can be met by Jabber’s ability to send messages to individual participants as well as to the whole group, but we still need to address the bulk transfer and ownership management requirements. By design, Jabber servers do not handle transfers of large amount of data between any two peers in order to provide efficient and fair routing for all users, hence, for simulation purposes, we have implemented a HTTP-based transfer using a web server. We propose a more efficient solution, a JXTA-based bulk transfer protocol that distributes the cost of uploading between all participants. The performance of this protocol is evaluated without integrating it with the Jabber-based protocols. Neither of the two frameworks readily provides the ownership management service and we propose to leverage the existing group notification service in conjunction with the presence service to implement a Jabber-based collaboration protocol to distribute ownership management over all participating users.

This paper is structured as follows. Section 2 places our application prototype in the context of related research and discusses an alternative approach; Sections 3 and 4 provide implementation details and experimental result for the collaboration and bulk transfer protocols respectively; and the paper concludes with Section 5.

2 Related Work

As part of a broad collaborative environments [12] research category, the present research can be compared with many tools and frameworks. Collaboration on multimedia presentations is described in [15], but it requires an application-specific TCP-based protocol and places the emphasis on the authoring tool that is subsequently interfaced with communication and concurrency control modules. Collaborative authoring is addressed in [7] which introduces a Web Distributed Authoring and Versioning protocol (WebDAV) extensions enabling shared awareness through event notification. A framework for collaborative applications over a JXTA is presented in [8] and it provides supplemental services such as synchronous message transport, room administration, peer communication support and application space management.

An extensive body of work discusses alternative solutions to Jabber-supported event notification. An infrastructure presented in [2] supports development of scalable applications composed of loosely-coupled services that communicate via XML document exchange. Authors report the service invocation overhead to be roughly equivalent to that of a light-weight RPC protocol, and over an order of magnitude less than XML-based protocols such as SOAP/HTTP. An application-level multicast framework proposed in [11] targets group communication in interactive applications such as real-time collaboration by specifying a stateless mechanism - application level header contains an encoding of IP addresses of all multicast group members - together with a corresponding tree building algorithm.

Bulk transfer is usually the domain of content distribution networks (CDNs). Recent studies suggest that users should share their upload bandwidth in order to help other participants get the file faster. Two technologies in particular, Application Layer Multicast (ALM) and implementation of a BitTorrent (BT) protocol make upload bandwidth utilization possible. ALM allows for multicasting data at the application as opposed to the network layer, and can be separated into end-host and infrastructure solutions. The end-host solution involves hosts sharing the responsibility for forwarding data to others by organizing themselves into an overlay tree [9]. The infrastructure solution assumes that the infrastructure itself does the heavy lifting of efficient multicast while individual members of the infrastructure also act as proxies for the participants [16].

A BitTorrent file distribution usually consists of an ordinary web server, a torrent file, an active and centralized component tracker, an initial client with the full copy of the file to share and a set of downloading clients. A new client starts by downloading a torrent file containing the IP address of the tracker from a website indexing such files. The tracker keeps information about the peers that are currently active and acts as a rendez-vous point for all the clients of the torrent. Active clients periodically report their state to the tracker or when joining or leaving the torrent. Upon joining the torrent, a new client receives from the tracker a list of randomly selected active peers to connect to. Clients involved in a torrent cooperate to replicate the file by exchanging chunks of the file with one another. BitTorrent specifies two main algorithms, the choke and rarest-first, to make the process of chunk exchange efficient [3]. The choke algorithm encourages cooperation among peers, limits the number of peers a client concurrently exchanges chunks with and selects the best peer (optimistic unchoke), while the rarest-first algorithm controls which pieces a client will actually request in the set of pieces cur-
that communicate by sending multicast messages. Ensuring the distributed system is treated as a collection of users made available to other users [13]. The implemented no- and how updates generated by one user are propagated and high responsiveness requirement from application usability. Perception of quality depends on the timeliness of response of the system to other users’ actions, whereas concurrent users’ actions cannot result in a globally inconsistent state, even in the presence of conflicts. Our group notification implementation focuses between two incompatible requirements [14], high responsiveness and high concurrency. Perception of quality depends on the timeliness of response of the system to other users’ actions, whereas concurrent users’ actions cannot result in a globally inconsistent state, even in the presence of conflicts. Our group notification implementation focuses on the high concurrency requirement, and we assess the high responsiveness requirement from application usability perspective. Group notification determines when, what, and how updates generated by one user are propagated and made available to other users [13]. The implemented notification mechanism conforms to the process group model - the distributed system is treated as a collection of users that communicate by sending multicast messages. Ensuring message delivery in the presence of delays and link failures is provided by the Jabber protocol implementation while our implementation addresses the issues of concurrent message ordering and changes to group membership. A notification policy determines when updates should be propagated, and depends on specific requirements of the application. In our prototype, users’ actions trigger an immediate update, if allowed by a locking mechanism. This locking mechanism is a type of floor-control technique [4], which should prevent conflicts from arising. In the rare case a conflict does arise from an attempt to obtain a lock, it is automatically resolved by our prototype.

The main collaborative functionalities in our application are provided by the Jabber Collaboration Interface (JCI) component. To avoid the inclusion of authoring semantics in JCI, a Document Object Model (DOM) is used to mediate between JCI and the authoring application. The JCI component has two main functions. First, it provides awareness of other participants’ actions and contributions by representing interaction between users in a collaborative space. This space is application-specific and is conceptualized by some collaborative space metaphor. Second, it ascertains that a particular placement of an object will not generate conflicts as defined by the DOM. The document object model is an XML file specifying the relation of multimedia objects to the collaborative space metaphor. For instance, an image will occupy a certain area at a certain position of a specific surface on a 3D object. For temporal objects, another variable would specify the duration of this relationship. DOM is the interface between the application and the JCI component by describing where and how conflicts may occur. For testing purposes a DOM that defines sixteen areas to which images are added is used, and as the end result of the collaboration, a collage of pictures is obtained. Finally, our application-specific code implements the collaborative space metaphor (single sheet of paper with predefined areas) and provides semantics for user actions such as multimedia object creation, modification and manipulation, which allows users to add, remove and order multimedia objects. Ordering can be spatial, temporal or both, as defined by the objects’ characteristics and collaboration goals.

3 Collaborative applications need to strike a balance between two incompatible requirements [14], high responsiveness and high concurrency. Perception of quality depends on the timeliness of response of the system to other users’ actions, whereas concurrent users’ actions cannot result in a globally inconsistent state, even in the presence of conflicts. Our group notification implementation focuses on the high concurrency requirement, and we assess the high responsiveness requirement from application usability perspective. Group notification determines when, what, and how updates generated by one user are propagated and made available to other users [13]. The implemented notification mechanism conforms to the process group model - the distributed system is treated as a collection of users that communicate by sending multicast messages. Ensuring message delivery in the presence of delays and link failures is provided by the Jabber protocol implementation while our implementation addresses the issues of concurrent message ordering and changes to group membership. A notification policy determines when updates should be propagated, and depends on specific requirements of the application. In our prototype, users’ actions trigger an immediate update, if allowed by a locking mechanism. This locking mechanism is a type of floor-control technique [4], which should prevent conflicts from arising. In the rare case a conflict does arise from an attempt to obtain a lock, it is automatically resolved by our prototype.

The main collaborative functionalities in our application are provided by the Jabber Collaboration Interface (JCI) component. To avoid the inclusion of authoring semantics in JCI, a Document Object Model (DOM) is used to mediate between JCI and the authoring application. The JCI component has two main functions. First, it provides awareness of other participants’ actions and contributions by representing interaction between users in a collaborative space. This space is application-specific and is conceptualized by some collaborative space metaphor. Second, it ascertains that a particular placement of an object will not generate conflicts as defined by the DOM. The document object model is an XML file specifying the relation of multimedia objects to the collaborative space metaphor. For instance, an image will occupy a certain area at a certain position of a specific surface on a 3D object. For temporal objects, another variable would specify the duration of this relationship. DOM is the interface between the application and the JCI component by describing where and how conflicts may occur. For testing purposes a DOM that defines sixteen areas to which images are added is used, and as the end result of the collaboration, a collage of pictures is obtained. Finally, our application-specific code implements the collaborative space metaphor (single sheet of paper with predefined areas) and provides semantics for user actions such as multimedia object creation, modification and manipulation, which allows users to add, remove and order multimedia objects. Ordering can be spatial, temporal or both, as defined by the objects’ characteristics and collaboration goals.

3.1 Collaboration Protocol

In order to maintain a consistent document during the authoring session, any modifications made in the collaborative space by a user must be coordinated with the actions of all other users. If a lock request is accepted by all the users, a modification may take place and becomes public after an unlock message. The collaboration protocol implemented in the JCI component conducts the locking and unlocking of spatial areas as defined in the DOM. The collaboration protocol uses time-stamps to resolve any conflicting lock requests. For each user, the JCI component maintains a list of locks that are currently in place for all users along with a list of pending requests that this particular user has made. Requests and replies are carried by group chat and individual messages respectively. There are five types of messages that the protocol defines and accepts. These simple XML-based message types are lock request, accept, deny, lock confirm, and unlock. All messages carry information about the issuer’s identification, time-stamp and area identification as specified in the DOM.

3.1.1 Time Synchronization

When two requests conflict, the protocol accepts the message with the earliest time-stamp. In order for the time-
stamp comparison to be relevant, it is necessary that users agree on a standard time. In order to get all the nodes to agree on time, we use a light time synchronization protocol based on Cristian’s algorithm [6]. One client node acts as time server, or leader. The leader sends out a series of values issued from the java call System.nanoTime(), whenever a user joins the conference room, or every two minutes; other nodes do not request them. Every user in the room, including the leader, takes whichever time-stamp came through within the shortest delay out of the series, and uses it as a reference when a time value is required.

3.1.2 Leader Election

In order for the nodes to obtain a reference time, a leader has to be elected. If a user joining the conference room is the first participant, then this user is designated as the leader. If there are other participants in the room, the newly joined user will wait for a leader declaration message. Upon receiving this message, the latest version of the DOM is retrieved and the collaboration continues. In the case where the leader leaves the room, the collaboration is terminated. A new leader is determined by selecting the user who has the highest hash code of their unique user name. Since each user has access to a list of participants in the room, this calculation returns the same result for all users. Once the new leader has been selected, the collaboration can continue once again.

3.1.3 Handling Group Changes

Group changes occur when a user joins or leaves the conference room and both events are handled similarly. Late join arises when a user joins a conference room that supports an on-going collaboration. This user then needs to obtain the latest DOM, which introduces two issues, obtaining the latest version of the DOM and ensuring that the DOM does not change while the new user is in the process of acquiring it. A file that meets both of those requirements is the authoritative version. Similarly, when a user leaves, its outstanding locks need to be deleted, and an authoritative version posted. Although all users have the same DOM and could therefore provide their copy, it makes conceptual sense to have the leader responsible for posting the authoritative version. When a user joins an empty conference room, it becomes the leader and retrieves and loads the DOM into memory. From then on, the leader is responsible for posting authoritative versions of the DOM. When a user joins a conference room that is not empty, it awaits a leader declaration message. Before the leader sends this message, it waits for all locks to be unlocked, while all other users stop issuing new requests (replies and time-outs are still processed). When all locks are unlocked, the leader posts the DOM and issues the leader declaration message. The new user then retrieves the authoritative version of DOM from the designated web server. During the course of collaboration and in the absence of any group changes, processing an unlock message will update the version of the DOM that the user has in memory. Unlock messages hence contain all necessary and sufficient details of the modifications made to the multimedia document.

3.1.4 Deadlocks

In both the leader election and group changes procedures, clients might have to wait for locks to be unlocked, which presents a potential deadlock problem. If a client shuts down or its connection to the Jabber server is severed (due to a failure or by design), the server detects that the TCP connection to that client has been closed, and assumes the client has left. The server then sends a presence notification to the other participants, which allows clients to discard all pending and active requests owned by the disconnected client. If, however, a client hangs (e.g. malicious client) and stops responding while the TCP connection is still alive, requests emitted by other peers would continuously time out. To handle this case, other participants would need to agree which client failed and requires an addition to our protocol. A second case when a deadlock can occur is when a client connects to the server and receives a message saying it has joined the group, but is not yet in the group when requesting the list of participants. This is an artifact of the Jabber server implementation found out during the development of our prototype and happens very rarely. Although not implemented, this case could be handled by ensuring that the client cannot proceed if it is not present on the conference room roster.

3.2 Setup and Experiments

We have simulated seven users collaborating over the Internet and the evaluation is based on the following four metrics. First, we measure the average time required to lock an area by subtracting the time at which a request was submitted from the time at which it was accepted by every collaborator, providing us with a lower bound for the delay in updating the multimedia document. Second, we measure the delay the user may experience while performing an addition to the multimedia document by taking the average total time to request a lock, obtain, it, upload a relevant object, and send out the update. Third, we count the proportion of non-successful requests which require the user to reconsider the change they’re trying to make and possibly redo it. Finally, we obtain those metrics when the size of the group varies from three to seven collaborators.

We have implemented a script that simulates users’ behavior by including the following four actions: joining a
chat room and downloading the latest version of the DOM, initiating changes to the document, updating the local copy when updates are received from other users and finally, saving a backup copy of the document whenever somebody leaves the conference room. Upon launching, our test program automatically logs in and joins a collaboration room hosted on a Jabber server. After joining, the script simulates a user who adds a new image every seven to ten seconds to an area taken at random among the 16 predefined areas of the document as specified by the DOM. Once the request is accepted and the lock confirmed, a CGI PHP script uploads a random picture to a common HTTP server. The picture is picked amongst a set of three pictures, with the following sizes and likelihoods of being chosen: Picture1, 35kB in size, 50% of the time, Picture2, 75kB in size, 33% of the time and Picture3, 200kB in size, 17% of the time. Once the picture is uploaded, the state of area is changed to reflect the addition of the new object and an unlock message is sent to peers. The following setup was used: the first client (User1) located in Gatineau (Québec), was connected to the Internet via a domestic DSL connection with a downstream capacity of 3.0 Mb/s and also hosted our HTTP server. The second client (User2) was connected to the Internet via the wireless network of the University of Ottawa. A home network of three computers connected to the Internet through a residential cable connection, hosted up to four clients: two PCs connected by wire (User3, User4 and User5); and a laptop connected over an 802.11b wireless connection (User7). Finally, a fourth computer hosting one user (User6) was connected to the Internet through a broadband cable connection in Ottawa. The Jabber server is hosted on jabber.org located in IOWA, US.

3.2.1 Request Delays

Fig. 1 shows the average of delays to obtain a lock, and total time to make a modification to the document, as a function of the number of users present in the collaboration room. Fig. 2 shows the per-request total time to make changes to the DOM throughout the life of the simulation, of around twenty-five minutes. It can be observed that the time to obtain a lock and to perform an update is relatively stable up to six users, with the majority of changes taking less than five seconds. The system requires up to three seconds on average (with seven participants) to confirm that the area is not locked by somebody else, and a few more to actually perform the change. The addition of User7 increases both the time to obtain a lock and to perform an update. We have identified three factors that might affect performance in our experiment. First, most public servers, in order to remain reliable and fair to all, use choking to delay or drop messages from a user who is sending heavily. Second, it is expected that our script attempts to upload, on average, around 10 kB/s worth of data, per user. This implies an average of 40 kB/s in upload for the network hosting User3, User4, User5 and User7. Taking into account that another bandwidth-consuming application was running on one of the machines (a BitTorrent client) it seems reasonable to believe that the upload bandwidth became saturated. Finally, with two of our test machines running wireless connections, it is suspected that this might cause some of the observed spikes in delays.

3.2.2 Request Results

The result of issued request is shown in Fig. 3. Requests not sent are those that were canceled either because the area was already locked, or because the activities were paused in order to let a user join the collaboration room. Canceled requests are requests that were retracted when another user was logging in. Timeout requests are those for which not
all accept messages were received within 5 seconds; those were re-issued. Denied requests are those that were withdrawn because another peer attempted to lock the same area a moment before. Successful requests are those that completed successfully. We observe a constant decline in the proportion of successful requests when more users participate. One of the reason is that our script attempts to make changes to a randomly chosen area. The probability of picking the same area increases with the increasing number of users, in fact in case of 7 users and if users made requests at the same time, around $1 - \frac{16!}{(16 - 7)!} / 16^7 \approx 78\%$ of requests would have at least one conflict. In real life, humans might act somewhat less randomly and make lesser amounts of conflicting updates. The increased traffic due to conflicting requests may explain the increased number of time-outs as well.

Results of a simulation where 7 peers download a file using CMS is presented in Fig. 4. The download completes when all the pieces of the original file are received. The graph shows the number of pieces received by each peer and the number of pieces transmitted by the source as a function of time, in units of 5 seconds. Since CMS allows to share a file only if it is complete, we observe that all participants connect to the only source offering the file. The communication between peers being TCP-based, the source divides its bandwidth more or less equally among all the peers. On the other hand, if the peers did not start at the same time we could expect different results.

We can improve this performance profile by extending CMS to be more suitable in the case where all participants start downloading at the same time, by removing the two restrictions of fully downloading a file before sharing it, and of sequentially downloading the file from the beginning. The resulting Extended CMS (eCMS) allows the peers to share individual chunks (64kB) from an arbitrary start, and in any order. This flexibility allows us to define two different ways of completing the download, as described in the subsequent sections. Compared to CMS, eCMS uses different content advertisements, provides support for new

**Figure 3. Results of issued requests**

**Figure 4. Number of pieces received and sent**

4 Bulk Data Transfer

In this section we address the bulk data transfer requirement of our collaborative multimedia application with JXTA protocols. JXTA is a set of protocols that specify basic functionality required by peer-to-peer type applications. Functions such as peer discovery, peer communication and their associated management are performed by publishing and exchanging XML advertisements and messages between peers. JXTA architecture provides a standard service discovery framework, allowing 3rd party services to be offered. CMS is an example of such a service.

The JXTA Content Management System (CMS) is a service used to manage and transfer shared files. Peers that wish to share a file with others generate a content advertisement. Peers that wish to download a specific content perform a search for an appropriate content advertisement and initiate the download. CMS implementation allows to download the same file from multiple sources, but requires the file to be fully downloaded before it can be offered to others (i.e. creating an advertisement for a partially-downloaded file is not supported). Although conceptually a request for the file is made, on a lower level, CMS handles requests and replies for the 64kB chunks that make up the whole file. Each request specifies the start and end of a range $(\text{chunk}_i, \text{chunk}_k)$ for $k > i$, which is sequential from the beginning of the file. A time-out is associated with each request, triggered if a hole has been found in the list of received chunks. When this happens, a request is issued for the missing range.

Results of a simulation where 7 peers download a file using CMS is presented in Fig. 4. The download completes when all the pieces of the original file are received. The graph shows the number of pieces received by each peer and the number of pieces transmitted by the source as a function of time, in units of 5 seconds. Since CMS allows to share a file only if it is complete, we observe that all participants connect to the only source offering the file. The communication between peers being TCP-based, the source divides its bandwidth more or less equally among all the peers. On the other hand, if the peers did not start at the same time we could expect different results.

We can improve this performance profile by extending CMS to be more suitable in the case where all participants start downloading at the same time, by removing the two restrictions of fully downloading a file before sharing it, and of sequentially downloading the file from the beginning. The resulting Extended CMS (eCMS) allows the peers to share individual chunks (64kB) from an arbitrary start, and in any order. This flexibility allows us to define two different ways of completing the download, as described in the subsequent sections. Compared to CMS, eCMS uses different content advertisements, provides support for new
messages, and its handling of issued and received requests for pieces (chunks) differs. As soon as a peer obtains a piece of the shared file, it will issue a content advertisement that contains the piece hash (for identification purposes) as well as the hash of the whole (parent) file. This ensures that peers interested in the file can discover other peers that are currently downloading that file as well. After joining such a swarm, peers will exchange state messages and attempt to download pieces from each other. The state message provides a way for peers to exchange information necessary to decide which pieces are available for upload. Once a peer has obtained all the pieces of the original file, it will create a content advertisement for the complete file and publish it at the RV peer. Other peers may however still make requests based on the previously discovered content advertisement for a piece. A message will be sent to the requesting peer notifying it to use the new advertisement instead. The above changes ensure that peers can share incomplete files with other peers. Each peer will keep track of the pieces it currently owns (successfully downloaded), and of the pieces it has issued a request for. Pieces that are not part of either set and are available at another peer may be included in a request for pieces. The piece selection algorithm, the format of the request and interpretation of the request are modified in the eCMS. In our implementation, pieces are selected randomly, or based on a previously signed partition. The format of the request is similar in both cases, and contains a sequence of numbers \((i, ..., k)\) identifying individual pieces. These in turn, are interpreted by the receiving peer as a series of requests for ranges \((\text{chunk}_i, \text{chunk}_{i+1}), ..., (\text{chunk}_k, \text{chunk}_{k+1})\) where \(i \neq k\). A time-out is associated with every piece, triggered if a piece has not been received within 20 seconds. That specific piece then becomes a candidate for another request.

### 4.1 Bulk Transfer Protocol

The purpose of the protocol is to allow peers to exchange information about pieces of a file, and to share them with other participants. In the next sections we describe how decisions are made about which piece to request, what messages are used to exchange information and pieces, and finally, the behavior of peers while exchanging messages.

#### 4.1.1 Sharing Pieces

We would like to maximize the availability of the pieces over all participants. This can be achieved by either peers randomly selecting which required pieces to request, or by the source assigning equal and disjoint file partitions from which peers must request required pieces. As the download progresses, there may be multiple peers offering the same piece, and each peer must keep track of the pieces it already owns, \(O_{\text{wn}}\{\ldots\}\), and of the pieces currently requested but not downloaded yet, \(D_{\text{downloading}}\{\ldots\}\). Each request for pieces is made from selecting some pieces from the set of remaining pieces, \(R_{\text{remain}}\{\ldots\}\) in order to create \(F_{\text{ull}}\{f_1, ..., f_{\text{full}}\}\). In the random case, a peer \(i\) will send a request for pieces to peer \(j\) by picking randomly up to 15 pieces from \(R_i\{\ldots\} \cap O_i\{\ldots\}\), where \(j\) can be another peer or the source. In the partition case, a peer \(i\) will send a request for pieces to the source by picking sequentially up to 15 pieces from \(R_i\{\ldots\} \cap P_i\{f_n, ..., f_m\}\), where \(P_i\{f_n, ..., f_m\}\) is the partition assigned to peer \(i\) by \(s\). If \(R_i\{\ldots\} \cap P_i\{f_n, ..., f_m\} = \emptyset\) then the peer picks sequentially up to 15 pieces from \(R_i\{\ldots\}\). If the other peer is not the source, then peer \(i\) sends a request for pieces to peer \(j\) by picking sequentially up to 15 pieces from \(R_i\{\ldots\} \cap O_j\{\ldots\}\). Partitions of equal size are assigned by the source peer on a first-come, first-served basis, and the source has prior knowledge of the number of participating peers.

#### 4.1.2 Exchanging Messages

There are four types of messages that participants exchange to complete the download:

**State Message.** A request for the state can be sent explicitly, or embedded in the request for pieces. Each peer will explicitly request state information from a peer it is currently communicating with whenever this state information is set to null. State information will be set to null at the initiation of the download, and whenever a request for pieces cannot be created due to lack of available pieces. This message is sent at the beginning and when selecting a suitable peer to download from. State request is also embedded (or piggybacked) in a request for pieces whenever the next request for pieces cannot be created due to lack of available pieces. This method of requesting state information is used whenever a peer is actively downloading from another peer. State request is composed of the message name, and in the case of an explicit state request, the requesting peer JXTA identification. State reply will be sent whenever a request for state is received. Each peer keeps a state bit array representing all the pieces of the file it currently can share with other peers. The maximum size of the bit array is less than 2400 bits, or 300 bytes (this is dependent on the size of the file to transfer and the size of the piece). This information, plus message name and responding peer JXTA identification form the payload of the state reply.

**Piece Message.** A request for pieces contains a numerical identification of requested pieces. Up to 15 pieces can be requested at a time, and the message contains the requesting peer JXTA identification. Replies to this request are handled by the underlying JXTA/CMS implementation, and are sent containing individual pieces (64kB chunks).
Complete Message. This message is rare, and its purpose is to notify that a peer has all the pieces. Other peers use it to stop sending state request messages to that peer.

Partition Message. In the partition case, two extra messages are exchanged between peers and the source only, the request and reply for a partition. These messages are exchanged prior to any piece requests. The purpose of these messages is to allocate a specific partition to each peer.

4.1.3 Peer Discovery and Behavior

A rendez-vous (RV) peer is used to facilitate the startup phase of the collaboration. All participants send a message to the RV peer advertising their presence (JXTA advertisement) and wait to discover 8 adverts (7 peers and 1 source) before proceeding with the actual file transfer. This ensures all peers start roughly at the same time. Only the source advertised that it has a file to share (CMS advertisement), and therefore all the peers attempt to connect to the source and request one piece of the file. Once this piece is received, a peer publishes its advertisement at the RV peer that it has a file to share. All the peers (except the source) attempt to discover new peers offering to share the file every 10 seconds. Once other sources of the file are discovered, peers start to behave as client and server with respect to each other.

Peers behave as client when requesting pieces from other participants, and as server when fulfilling requests for pieces from other participants. Server behavior is provided by the underlying JXTA/CMS implementation (with the slight modification to handle state messages), and is therefore the same for all participants. Since the source has the complete file, it never behaves as a client and it never receives requests for its state information. Other participants implement exactly the same client behavior. They select up to 4 peers to connect to, chosen from a randomized list of discovered peers. A connected peer is dropped in favor of another peer if it fails to upload pieces in three attempts, either by not having any pieces to upload or if a requested piece is not received within 20 seconds. Upon receiving all the pieces to reconstruct the original file, a client updates its content advertisement and becomes a source.

4.2 Setup and Experiments

As mentioned previously, the performance of this protocol is evaluated without integrating it with the Jabber-based protocols. The setup consists of 3 bridged LANs, the first providing 100Mbps supports 4 PCs (a.k.a fast peers), the source (peer with the complete file) and the RV peer, the second providing 100Mbps supports 1 fast peer, while the last providing 10Mbps supports 2 PCs (a.k.a slow peers). We ran 5 tests for both the random and partition cases and we collected logs for each machine after each test. We use arithmetic averages over those tests in all cases, unless otherwise specified. The size of the transferred file is 152.5MB, each piece is 64kB, and thus we have 2383 pieces.

4.2.1 Time Required to Complete Download

The time necessary to complete the download is a primary metric. The download completes when all the pieces of the original file are received. Instead of showing the cumulative number of pieces received as a function of time, the graphs in Fig. 5 show the number of pieces received by each peer as a function of time, in units of 5 seconds. Peers request pieces based on the information about which pieces a participant can provide. This information is obtained by requesting a state message. The number of state messages sent as a function of time is shown on the graphs as well.

![Figure 5. Time to complete download](image-url)
same in both the random and partition cases, although most of the fast peers seem to finish around 20 seconds faster in the random case than in the partition case. Slow peers on the other hand seem to finish around 20 seconds faster in the partition case than in the random case. We have also ran test with only the source and one fast peer, yielding 97 and 81 seconds for random and partition cases respectively. Due to the limited nature of the experiments however, no definitive inference can be drawn, and further study is necessary. In both cases, peers start by downloading from the source, and around 30 seconds later obtain advertisements that allow them to start sharing pieces. This is an artifact of our implementation, where JXTA discovery happens periodically every 10 seconds. The period when peers download only from the source could be reduced by reducing this discovery period. The number of packets peers can obtain from the source will be a function of their own connection speed, and of how many peers the source is serving. After a peer discovers other sources of pieces, the number of pieces received increases for the fast peers and drops for the slow peers. This is expected since fast peers have more download bandwidth than is available to them from the source only, whereas slow peers must now share their limited bandwidth with other peers. In this case, slow peers cannot improve their performance by connecting to multiple peers, and should limit their download connections to less than 4 currently allowed in the simulation. In order to minimize downloading time for all participants whenever there is more fast peers relative to slow peers, slow peers should generally behave as clients only, although the exact balance for which this holds needs further study.

4.2.2 Load on the Source and Users

We also look at the sustained load by the source and peers, expressed as the number of pieces transmitted as a function of time, in units of 5 seconds. The sum of the peers’ contribution is shown instead of the load on individual peers. This gives us a comparison of the work performed by the source relative to the work performed by other participants and is shown in Fig. 6.

![Random Load](image)

![Partition Load](image)

Figure 6. Load on the source and users

4.2.3 Cost of Sharing

Next, we assess whether there is a relationship between providing pieces (behaving as a server with respect to other peers) and download time (time to receive all the required pieces). The scatter diagram in Fig. 7 presents results for each trial (averages are not used), plotting download time as a function of number of pieces uploaded. To highlight the relationship, linear and quadratic fitting is used for slow and fast peers respectively.

Due to the wide bandwidth difference, slow peers fulfilling requests from fast peers will have a marginal effect on the fast peer’s performance, but at a substantial cost to themselves. This is clearly shown in the diagram where increasing number of shared pieces increases the time necessary to complete the download. The trend for fast peers is more subtle, and shows a slight decrease of download time as the number of pieces transmitted increases, up to around 2500 pieces where the trend flattens (download time remains constant with increasing number of pieces). The initial decrease can be explained by the fact that by providing less pieces a peer removes upload bandwidth from the system, which increases the download time for all participants. Similarly, the flattening of the curve is consistent with the fact that once a peer finishes downloading the entire file, it starts to behave as a source to other peers. Finally, comparing the random and partition cases reveals broadly the same trends. If we design for the smallest download
time for all peers, this would imply that the fast peers will share at least around 2500 pieces (close to the size of the file to share), while the slow peers should not share at all. This optimal design will most probably change with different proportions of fast to slow peers, as well as increasing heterogeneity of peers’ available connection speed.

5 Conclusions

A collaborative authoring application will require certain basic services which can be accessed through Jabber and JXTA protocols. Jabber was designed to provide near real-time structured information exchange and thus naturally provides the required group notification service. We are using this service, in conjunction with the presence service to create a collaboration protocol by distributing the responsibility for document consistency to participating users. The implementation of this protocol mandates a strict document consistency, and presents a great scope for improvements by taking the application semantics into consideration. Future work includes running the experiments on a single high-bandwidth network and to reduce the probability of conflicting requests in order to obtain a measure of the protocol’s performance while in a controlled environment. Finally, JXTA framework was designed to facilitate communication and collaboration between peers, and thus lends itself to provide the bulk data transfer service. Future work will address the following two issues. In case of heterogeneity of peers’ connection speed, better results can be achieved if certain peers do not share their bandwidth, or limit their upload rate. This suggests that deriving bounds for this optimum point and providing ways of achieving it, need to be explored. Secondly, we have highlighted two approaches that - by varying the way in which the content is downloaded from the original source - impacts participants’ performance. Further study is needed to determine which approach presents better characteristics, and if they are specific to particular cases. Additionally, a comparison of our multimedia authoring implementation to an implementation based on traditional web services standards could yield further insights.

References