Opportunistic Information Distribution in Challenged Networks

R. Metzger, M. Chuah
Department of Computer Science & Engineering
Lehigh University
rcm2@lehigh.edu, chuah@cse.lehigh.edu

***Abstract*—Mobile nodes in some challenging network scenarios suffer from intermittent connectivity and frequent partitions e.g. battlefield and disaster recovery scenarios. Disruption Tolerant Network (DTN) technologies are designed to enable nodes in such environments to communicate with one another. Several DTN routing schemes have been proposed. In order for the DTN technology to be useful, enhancements need to be made to legacy applications e.g. emails, web browsers, instant messaging etc to enable such applications to work over DTNs. In this paper, we present a DTN Jabber proxy which we have developed to allow mobile devices to run Jabber applications over DTN. Our proxy can identify group chat messages which enable our prototype to perform last mile multicast to minimize wireless bandwidth usage.**

***Index Terms*—disruption tolerant networking, DTN, legacy application, Jabber, Mobile Ad Hoc Networking.**

I. Introduction

With the advancement in technology, many users carry wireless computing devices e.g. PDAs, cell-phones etc. Such devices can form mobile ad hoc networks and communicate with one another via the help of intermediate nodes. Such ad hoc networks are very useful in several scenarios e.g. battlefield operations, vehicular ad hoc networks and disaster response scenarios. Many ad hoc routing schemes have been designed for ad hoc networks but such routing schemes are not useful in some challenging network scenarios where the nodes have intermittent connectivity and suffer frequent partitioning. Recently, disruption tolerant network technologies [1],[2] have been proposed to allow nodes in such extreme networking environment to communicate with one another. Several DTN routing schemes e.g. [3],[4] have been proposed.

Although routing is an important design issue for such sparsely connected networks, it is important to ensure that popular legacy applications e.g. emails, web browser, instant messaging can operate over challenged networks that run DTN. Several projects have studied email over DTN e.g. in the context of military networks [5], nomadic community networks [6] and heterogeneous environments [7]. The first two approaches are proxy-based approaches but such approaches do not exploit the opportunistic communication opportunities available between mobile nodes. In [7], DTN gateways are deployed to store mail messages destined to a server/client on the Internet and intermediate DTN nodes can store mail messages destined to recipients which are mobile DTN nodes for forwarding in the future.

Apart from email, the instant text messaging application is another popular application that many mobile users often use. There are multiple open-source text messaging applications e.g. Chat, Jabber. In this paper, we describe a Jabber proxy that we develop to allow mobile devices to run Jabber application over DTN. Our Jabber proxy provides several functions: (a) redirects the Jabber traffic through the DTN stack, (b) ensure that the internal timeout timers inherent to TCP do not expire, (c) parses connect/disconnect messages to allow mobile nodes to join and leave at will, and (d) parsing of group chat messages such that last mile multicast transmission can be used to reduce wireless bandwidth usage.

The rest of the paper is organized as follows. In Section II, we discuss related work. In Section III, we describe our Jabber proxy design. In Section IV, we describe a 12-node testbed that we have built to evaluate our Jabber proxy prototype. We also discuss the current limitations of our prototype system. We conclude in Section V by discussing near future work that we plan to explore.

II. RELATED WORK

In [5], the authors develop both an email and a web proxy. The email proxy generates appropriate SMTP responses to cause the client to perform its tasks. The proxy then uses some other mechanism, such as DTN to ship information about the application layer action to a peer proxy. In [7], the authors present a prototype system that can deliver emails within a DTN network, from a DTN network to the Internet, and from the Internet to a DTN network. Their proposed architecture can integrate traditional and DTN-based mail delivery.

III. OVERVIEW OF BASIC JABBER PROTOCOL

In Figure 1, we show a simple network with two clients and two servers. The two server are named Server 1 and Server 2. Each client is associated with a server. Each client has a client Jabber ID (JID) which is composed of a hostname, the symbol '@', and the name of the server it is associated with. The JID can also have an optional resource tag that allows a particular user to log in from multiple

locations simultaneously with different resource tags. Thus, a JID has the following form node@server[/resource]. The clients in our scenario will have the JIDs client1@server1 and client2@server2.

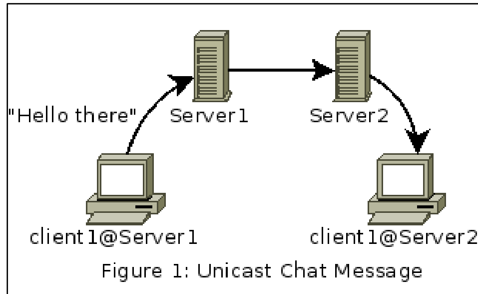


Figure 1: Unicast Chat Message

A. Basic Jabber Features

There are many features in the Jabber Protocol. Here, we merely present those features that are relevant to our testbed demos. For more information, the readers are referred to RFC3920[8], RFC3921[9], RFC2222 [10], and associated RFCs and Jabber Extension Protocols [11]. Before a user can use any chat feature, the user must first log into his/her respective server. The Jabber Client login message sequence is shown in Figure 2.

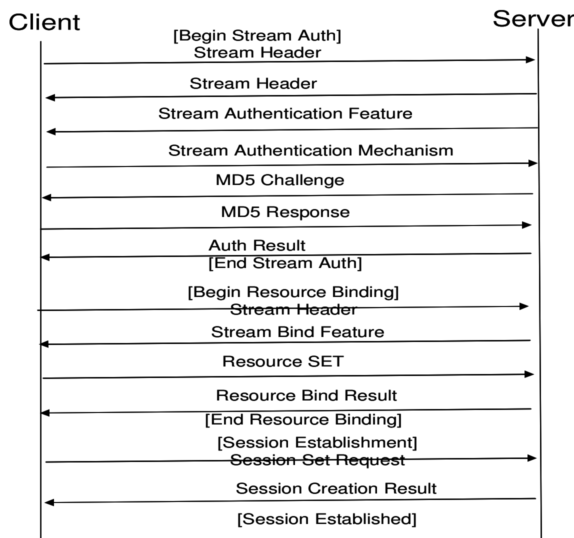


Figure 2: Jabber Client Login Message Sequence [8]

Once both clients have logged into their respective servers, the Jabber system is in a stable state. This changes when client1@server1 tries to send a message to client1@server2 through its established TCP session with server1. The basic format of this message is as shown in Figure 3.

```
<message type="chat" to="client1@server2/Home"
from="client1@server1/Home">
  <body>
              Hello there
  </body>
</message>
```

Figure 3: Jabber Message Content

The example in Figure 3 has tabs and new lines added for the clarity of the presentation, but this white space is not present in the actual transmitted message. There are several child tags of the message and message tag parameters that *may* be present. When this message is sent from client1@server1, it will be sent over the established TCP session to Server1. From the content in the "to" field, Server1 sees that it is not the destination server. Thus, it extracts the destination server from the "to" parameter and attempt to send the message to Server2. Since there is no established session between Server1 and Server2, Server1 will first use the server dialback mechanism to establish a session with Server 2. This will proceed similar to what is shown in Figure 2, except the stream authentication will use the server dialback mechanism instead of MD5 handshake. The server dialback mechanism is detailed in Figure 4.

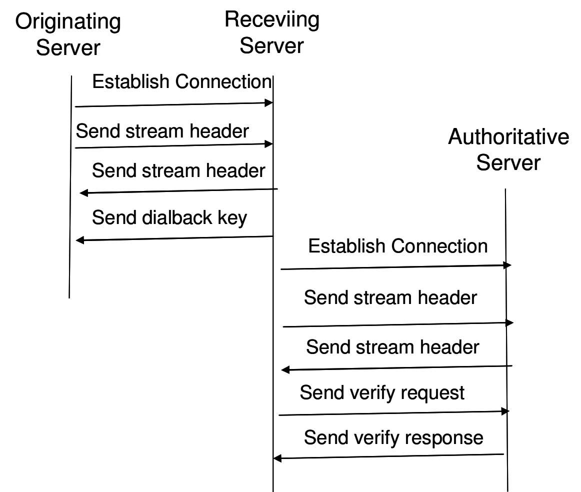


Figure 4: Sever Dialback Mechanism [8]

All messages from the Originating Server (OS) to the Receiving Server (RS) go over a single TCP connection that persists as long as the session from the OS to RS is alive. This TCP connection is from a random port at the OS to port 5269 at the RS. Then, RS connects to the Authoritative Server (AS); this is done over a separate TCP connection that is created for the verification step, and destroyed after the verification step has concluded. This second TCP connection is from a random port at the RS to port 5269 at the AS. This means that during the verification step, there are simultaneously 2 TCP connections, one from the OS to the RS, and another from the RS to the AS. Since in our case, the AS is the same as the OS, these 2 TCP connections are between the same two machines. Once this process is completed, there is a Jabber session from the OS to the RS but there is NOT a session from the RS to the OS. If client1@server2 wants to send a message to client1@server1, then Server2 will have to repeat this same process to create a Jabber session with Server1, but in this case Server2 will be the OS, and Server1 will be the RS. During the setup of the server, one needs to specify the DNS name of the server, the dialback process at both sides resolves the DNS names of each other. The resulting IPs

are used as seeds for creating the keys used in the dialback sequence. This of course means that the specified DNS names must be resolvable, and must resolve to the SAME IP by all involved parties.

Once the Jabber session from Server1 to Server2 is established, Server1 will forward the message sent by client1@server1 to Server2. Server2 will examine the "to" parameter of the message and see that it is the destination server. Server2 will then check to see if client1@server2 is logged in. If the client1@server2 is logged in, then the message is sent to the client. Otherwise, the message is put into persistent storage to be delivered when the client logs in.

Every jabber exchange is two streaming xml documents, one in each direction. Thus, if we take every message from the beginning to the end of a Jabber session that goes in a particular direction, it forms a valid xml document. This is why the Jabber protocol is sometimes referred to as a streaming xml protocol. The "</stream:stream>" tag is the closing tag for this streamed xml document. Figure 5 shows the streaming XML structure within a Jabber session. There can be as many message and iq tags as needed within a session. The stream tags enclose the session. If a crash causes the stream to not be properly closed, the system relies on the timeout of the underlying TCP socket to force the other side of the stream to terminate.

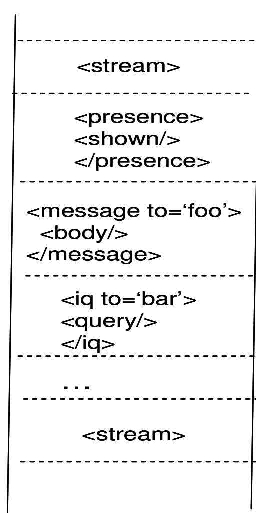


Figure 5: Streaming XML Structure [8]

B. Group Chat Features

The servers are equipped with a multicast plugin that provides a group chat feature. Using the same scenario described earlier, once all sessions are established, the clients can use these sessions to send a special iq message to chat_room1@conference.server2. The special name conference.server2 within the iq message allows the message to be "routed" to the chatroom plugin on server2. This iq message allows the sending client to join the chatroom named chat_room1 on server2. If a second client client2@server2 was to establish a session with Server2 and join the chatroom, then, there would be 3 users in the chatroom. If client1@server1 wants to send a message to the chatroom, the flow of messages will be as shown in Figure 6.

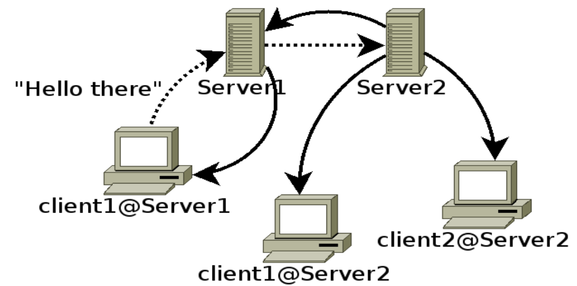


Figure 6: Unicast Groupchat Message

The dotted lines indicate the message heading towards the chatroom and the solid lines are the messages from the chatroom to each client that has joined that chatroom. We notice from Figure 6 that each server has to duplicate as many messages as there are registered users that join that group chat channel e.g. Server2 has to send duplicate messages, one to client1@server2 and one to client2@server2. If the two users are within the same transmission range from the server, then the server needs to multicast the message once. As the number of users registered with a particular server grows, this multicast optimization allows us to save energy and bandwidth for wireless devices that run Jabber program.

IV. DTN Jabber Proxy Design

Before we describe our proxy design, we first discuss the Jabber protocol details that affect our proxy design.

A. Jabber Protocol Details that Affect Proxy Design

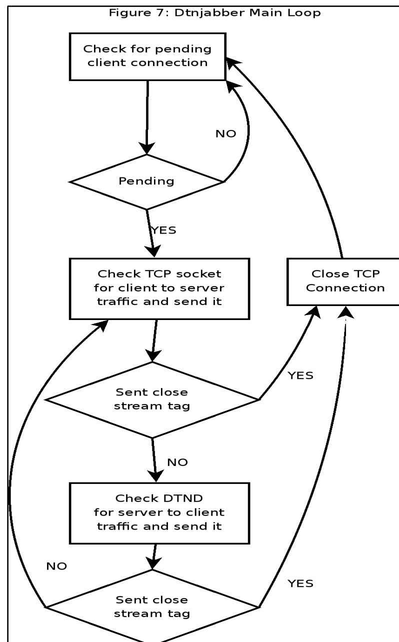
The first step in development of the proxy was to identify the ways in which the various jabber components use the network, so that the proxy can be designed to support such Jabber operations over a DTN. Our first observation is that once the login session is over, the sending of any chat messages does not result in any application layer response. All messages are sent via TCP packets so it suffices for the proxy to ensure that the sender receives a TCP acknowledgement back in time to prevent any TCP retransmission. The second thing we observe is that the Jabber components employ heartbeat messages, each consisting of a single space character (' ') on all TCP sessions. These are NOT part of the jabber protocol specification. Such messages are merely used to keep the TCP sockets from timing out during long period of inactivity and that jabber application does not do anything with these packets except to ignore them as invalid messages. Next, we observe that security settings are required for the server to server dialback mechanism to work. To support this mechanism, a server may have two or more established TCP sessions – one to each server it attempts to communicate with and one to the authorization server during the verification step. Finally, the jabber proxy needs to process close stream messages immediately since

the established connection needs to be torn down upon proper termination of a session e.g. the connection related to the verification step. As mentioned earlier, the close stream message is simply the "</stream:stream>" xml tag.

B. DTN Jabber Proxy Design

The DTN Jabber Proxy consists of two modules, one that runs near a jabber client, dtnjabber, and another which runs near a jabber server, dtnjabberd. Our proxy design allows a proxy module to be run on a separate machine from the one that runs the Jabber components. However, in our testbed set up, the proxy module runs on the same machine that runs the Jabber components since our current implementation does not handle a disconnect between the jabber component and its associated proxy component yet. This feature is left for future work. All sockets created by our proxies are non-blocking to avoid having to create a multi-threaded dtnjabberd or dtnjabber.

The dtnjabber proxy component is fairly simple, since it only supports one client. Thus, it doesn't need to maintain a table of connections. It simply needs to keep track of whether the associated jabber client is connected or disconnected. Figure 7 is a diagram that illustrates the state machine of the main loop of the dtnjabber proxy that runs at the Jabber client side.



The jabber client must be configured to send its traffic to dtnjabber instead of directly to the jabber server it wants to sign into. In all client software packages that we use, this is done easily via the "connect to server and port" configuring parameters. Then, dtnjabber simply opens a listening TCP port (the port number is specified via a command line argument) and waits for the jabber client to connect to that port. This causes the client TCP session to be terminated locally, so that in the case of a disconnect between the proxy components, the local TCP session will not time out while waiting for a TCP ACK packet. Also specified at the command line of dtnjabber is the EID of the dtnjabberd that is associated with the jabber server that the jabber client attempts to log into. Once a Jabber session is established, the dtnjabber proxy will look for a "</stream:stream>" tag. When the proxy sees this closing tag, it will forward the message and then close the link.

The server-side proxy, dtnjabberd proxy, must be designed to scale since it needs to keep track of numerous connections simultaneously. The dtnjabberd proxy maintains two connection tables. When a dtnjabberd proxy is invoked, it reads a configuration file that contains a list of servers that this particular server (say Server2) can connect to. For each remote server listed in the configuration file, a listening port is created and the listening port number is captured in a connection table referred to as Table 1. Table 1 captures all the outgoing connections initiated from Server 2. Each entry in Table 1 contains the following information: (i) TCP outgoing connection socket, (ii) an EID to which to send information received on that socket, and (iii) the last time there was traffic on the TCP socket. Currently, two outgoing connections for each remote server is supported. This design decision is made to support the server-to-server dialback mechanism (shown in Fig 4) where two TCP connections are launched for each server that Server2 wants to communicate with. One connection is for the authorization purpose. Our current design does not remove the connection entry associated with the authorization server even after that connection goes away upon completion. Table 1 is an example of what this table would look like on Server2 in our original unicast chat scenario shown in Figure 1.

	listen socket	Conn. socket	Flg	Timestamp
Server1 Jabber Connection	15001	56	1	
Server1 Auth. Connection *	15001	58	2	

- In our testbed setup, Server2 acts as the Authorization Server for Server 1.

Table 1: Outgoing Server Connection Table

The proxy will insert a flag byte in all the outgoing bundles from Server2 to a remote server (Server1 in our example). The flag value is set according to whether the established connection is the first or second outgoing connection to that remote server. The remote server will insert a flag value of 3 in all bundles sent in the reverse direction that contain responses to the first outgoing connection. A flag value of 4 will be used by the remote server for bundles sent in the reverse direction for the second outgoing connection. Note that once the session between two servers is established, the

data bundles often flow in one direction e.g. from Server2 to Server1 or from Server1 to Server 2.

A second connection table called Table 2 is used to maintain information related to incoming connections from remote servers and clients. The maximum table size of Table 2 is hardcoded in the #define of OPENCON line at the beginning of our proxy code. The entries in Table 2 are generated dynamically by the incoming DTN traffic. When an incoming bundle enters dtnjabberd, its source EID and flag are checked against entries in Table 2. If there is a match, then the bundle payload (after removing the flag byte) is sent to the TCP socket of that entry. If there is no match, then, an entry is created and a new TCP session is established with the current server (Server2 in our example). This newly created entry contains the following information: the source EID (which includes application registration name) of the incoming bundle, the TCP socket of the newly created TCP session, and the flag value of the incoming bundle. Once the new entry is inserted, and the TCP session is created, then, the bundle payload (after removing the flag byte) is sent to the new TCP socket. An example of how this table looks like on server2 after the initial session establishments for the scenario in Figure 1 is shown in Table 2.

source EID (c string)	connection socket	flag
dtn://client1.dtn/jabber	90	5
dtn://server1.dtn/jabberd	95	1

Table 2: Incoming Connections from remote servers

Our proxy inserts a one byte flag at the beginning of every bundle payload sent between proxy components. Currently, the flag value is used to distinguish bundles that arrive from different incoming connections between two jabber servers (for supporting the dial back mechanism). The flag byte is inserted because source EID alone is not sufficient to distinguish different incoming connections (recall that we need two separate connections from the same server because the same machine is used as the authorization server in our setup). The flag value is simply an integer between 0 and 255. A flag of zero is reserved for future use. The flag is set to one for all traffic sent via the first outgoing server to server session, and to two if it is the second session. A flag of three is used for bundles that contain response messages to the first server-to-server connection set up by a remote server. Similarly, a flag of four is used for the second server to server session created remotely by that same remote server. Finally, a flag value of five is attached to any bundles sent in any server to client or client to server connection. In future versions, a flag value of greater then 5 may be used to allow multiple clients to connect to a single dtnjabber instance to differentiate the traffic to and from the different clients. In our current dtnjabber version, only one jabber client may associate with a dtnjabber instance so a flag value greater than five is not used. When a Jabber session ends, the dtnjabberd proxy will receive a bundle containing a closing tag. The dtnjabberd proxy forwards this bundle, and then deletes the associated connection entry from its connection table. The main loop of the dtnjabberd program is detailed in Figure 8.

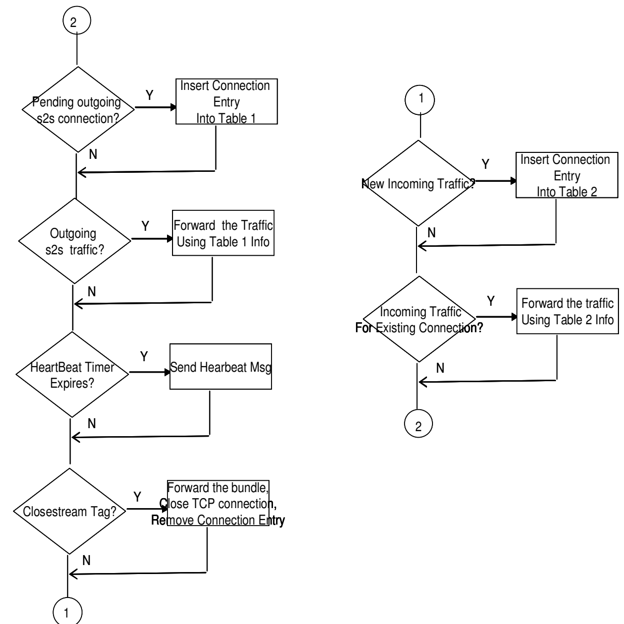


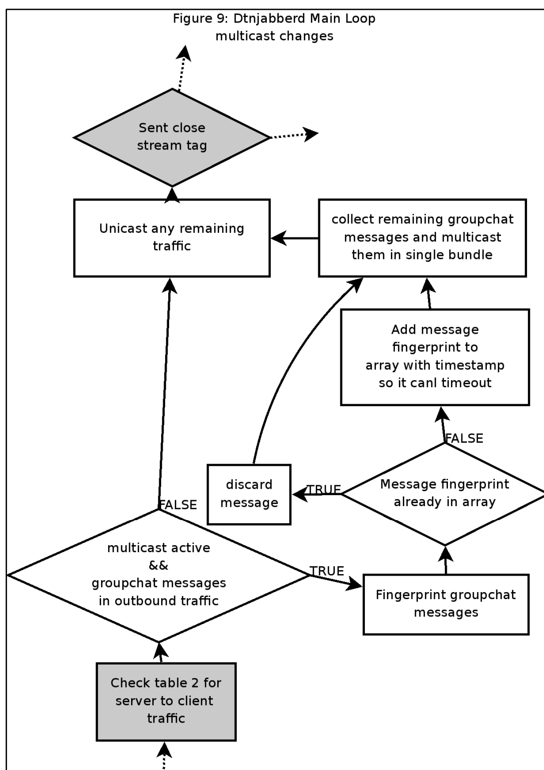
Figure 8: Main Loop of DTNJabberd

C. *Multicast Extension*

There is a MultiUser Chatroom (MUC) plugin in the Openfire Jabber server package that we use. This MUC plugin allows users to enter into a chat room and talk to one another. The default behavior of this plugin is that whenever a server receives a message for a group chat channel, it replicates that message to every user who logs into the chatroom. Currently, even if all the users are within the same transmission range from the server, the server will still send unicast messages to all of them. Since wireless communication is broadcast in nature, we add an integrated proxy and DTN2 feature to support multicasting on shared medium so that the server uses fewer transmissions. To accomplish this, 3 new functions need to be added to the proxy components. The first function is to identify messages to be multicast within dtnjabberd. Since jabber uses xml, and all message tags have a source field, this is as easy as identifying those message tags which have the type parameter set to group chat, which indicates that the message originates from the MUC plugin. We accomplish this using the GNU regular expression support in glibc, and performing a regular expression match against outgoing server to client payloads. Once identified, the messages are separated from the rest of the payload and sent via multicast; the remainder of the payload can still be sent normally via unicast. These modifications result in the changes in the dtnjabberd flowchart detailed in Figure 9, with the original functions shown in grey. The source EID of such multicast bundles is changed to a multicast EID so that the DTN layer can broadcast this message.

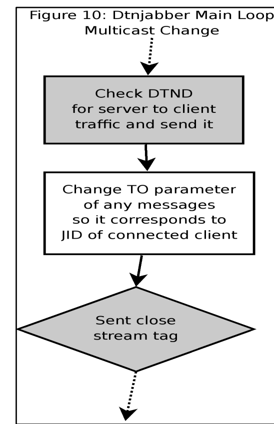
The second function that must be added is related to how incoming multicast bundles will be handled in dtnjabber. One of the other fields that the message tag contains is a "to" field which is the destination jabber ID that the message is destined for. Since we change the destination ID of group chat messages is changed to a multicast EID, dtnjabber at the receiving side needs to change it back to a client Jabber ID at the receiving end so that that multicast bundle can be passed to the Jabber layer . These modifications result in the changes in the dtnjabber flowchart detailed in Figure 10, with original functions shown in grey.

The last function that must be added is the pruning of duplicate messages, which is shown in Figure 9. In the current Jabber implementation, a server will generate as many messages as there are users associated with it even if



all these users are within the same transmission range. Since we cannot modify the Jabber server software, we have to insert this pruning function to remove redundant messages that need to be multicast. The complicating factor in this is that with even moderate traffic there is no guarantee of the order of packets being sent from the MUC plugin. Thus, duplicate message detection will be done by generating a hash of the body of all outgoing multicast messages. This hash will be compared to an array of hashes. If there is no match, the hash is added to the array for a period of several seconds (since duplicate messages shouldn't be far behind). If there is a match, then the message is a duplicate, and will be discarded.

At the moment, we rely on configuration at the network layer to let the server node knows which JIDs are within its transmission range.



V. TESTBED EVALUATIONS

A. *Testbed Descriptions*

We have set up a 12-node testbed as shown in Fig 11 to evaluate both our Jabber proxy, and the last mile multicast feature we have implemented. The forwarding nodes, R1 to R4, are robust wireless routers which have one active wireless LAN interface. The two Jabber server nodes, S1 and S2, are Dell laptops that have two wireless interfaces. C11, C12 and C13 are also Dell laptops. C21, C22, and C23 are Nokia Internet Tablets N770. All devices run Ubuntu and enhanced version of the DTN2 reference implementation [12]. The enhancement includes our Jabber proxy modules, enhanced Prophet module, and the last-mile multicast extension. We use an enhanced Prophet scheme as our DTN routing scheme. Our enhancements include some changes to remove routing loops and removal of several bugs that are present in the released Prophet code. We use Openfire Jabber server package. The Jabber clients composed of a mixture of Gaim, Gajim, and the jabber client included in the Internet Tablet OS 2006 on the Nokia 770 devices.

B. *TestBed Evaluations*

In our experiment, there are two group chat channels. When a text message is sent by a client, C11, to the group chat channel 1, S2 will send a copy to S1. S1 will multicast the group chat message so that clients C12, and C13 can receive it. Similarly, when a client C22 sends a message to the group chat channel 2, S1 will forward a copy to S2. S2 will multicast so that clients C22, and C23 can receive it. Currently, there is no support for dynamic group membership yet because the source code for Openfire Jabber server is not available. Forwarding nodes are configured with the appropriate multicast EIDs so they will re-broadcast traffic received from those multicast EIDs.

IP Filters are used to create the connectivity topology as shown in Figure 11. In addition, we emulate intermittent connectivity using IPfilter such that a link between two nodes may periodically go up and down. Figure 12(a) shows the screen dump we obtain at the node C11 during the process when the nodes C12 and C13 join after C11 joins. Figure 12(b) shows the screendump obtained at the node C11 during the process when the 3 nodes exchange

text messages with one another. During our testing, we notice that due to its slower CPU and limited memory, a Nokia 770 device can easily be overwhelmed by the multiple bundles created by the Prophet module and sometimes hangs. The laptops have no such problems at all. The DTN2 reference implementation is not implemented very efficiently and hence it does not work well when it is run in mobile devices with limited resources.

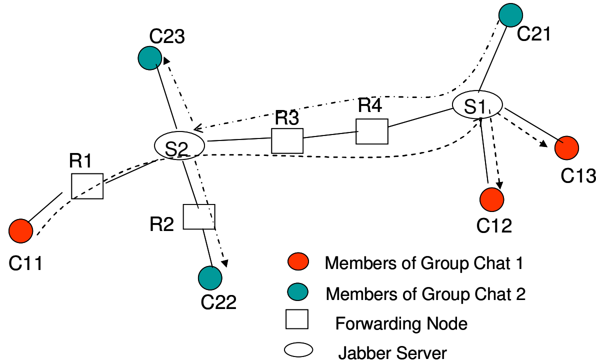
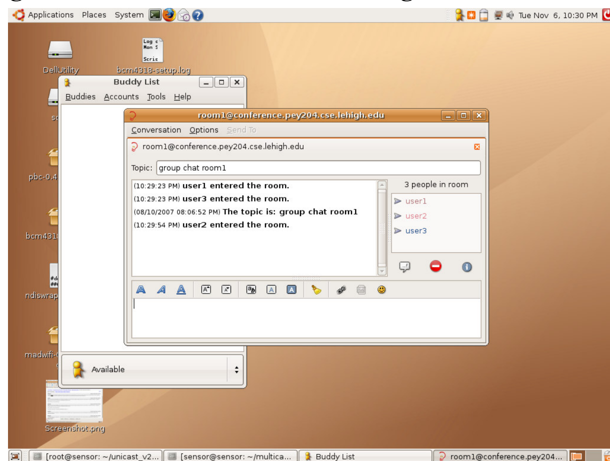
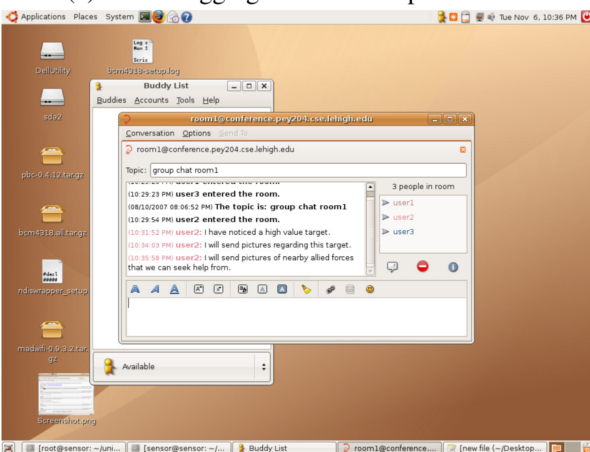


Figure 11: Testbed for demonstrating multicast feature



(a) Users Logging into the Group Chat Room



(b) Users in the Group Chat Room exchanges messages with one another

Figure 12: Screen Dumps at Node C11

C. *Limitation of Current Prototype*

Our current prototype does not support management of dynamic membership. In addition, we only support last-mile multicast. We are now developing new features where users can dynamically send a join message to a custodian node to join a multicast group, and a multicast source can send multicast packets to a custodian node. The custodian node will distribute the multicast packets. We also intend to develop a multicast routing module. A technical report describing these new features will be released in the future.

VI. Concluding Remarks

In this paper, we first give a high level overview of how Jabber messaging system works. Then, we give a description of how our DTN Jabber proxy works. Our proxy supports various functions including parsing the messages to look for group chat messages. Last but not least, we present a description of a 12-node testbed that we have built to evaluate our Jabber proxy software. We evaluated a scenario where users joining a Jabber group chat room can successfully receive group chat messages despite intermittent connectivity. Our prototype system currently does not support dynamic joining of group members. We intend to add this feature in the near future.

Acknowledgment

We thanked P. Yang for helping to extend DTN2 software with the last-mile multicast feature. This work has been supported by DARPA under Contract W15P7T-06-C-P430. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsor of this work.

References

[1] K. Fall, “ A delay tolerant network architecture for challengednetworks”, Proceedings of ACM Sigcomm, 2003..
[2] V. Cerf et al, “Delay Tolerant Networking Architecture”, RFC4838, April, 2007
[3] A. Lingren et al, “Probabilistic Routing in Intermittently Connected Networks”, Proceedings of Workshop on Service Assurance with Partial and Intermittent Resources, Aug, 2004.
[4] J. Burgess et al, “MaxProp: Routing for vehicle-based disruption tolerant networks”, Proceedings of IEEE Infocom, 2006.
[5] K. Scott “Disruption tolerant networking Proxies for On-the-Move Tactical Networks”, Proceedings of IEEE Milcom, pp 3226-3231, 2005
[6] A. Doria et al, “Providing Connectivity to the Saami Nomadic Community”, Proceedings of 2nd International Conference on Open Collaborative Designfor Sustainable Innovation, 2002.
[7] T. Hyyrylainen et al, “Opportunistic Email Distribution and Access in Challenged Heterogenous Environments”, Proceedings of ACM CHANT, Sept, 2007.
[8] P. Saint-Andre, Ed. “Extensible Message and Presence Protocol (XMPP): Core”, RFC3920, Oct 2004
[9] P. Saint-Andre, Ed., “Extensible Message and Presence Protocol (XMPP): Instant Messaging and Presence”, RFC3921, Oct, 2004
[10] J. Myers, “Simple Authentication and Security Layer (SASL)”, Oct, 1997
[11] XMPP Extensions, http://www.xmpp.org/extensions/
[12] DTN2 Reference Implementation, http://dtnrg.org/wiki/Code.