A Generic Multicast Transport Service to Support Disconnected Operation*

Silvano Maffeis
maffeis@acm.org

Walter Bischofberger
bischi@ubilab.ubs.ch

Kai-Uwe Mätzel
maetzel@ubilab.ubs.ch

Department of Computer Science, Cornell University, and UBILAB, Union Bank of Switzerland

Abstract

Many mobile computing applications can profit from process groups and reliable multicast communication to maintain replicated data, but most operating systems available today fail in providing the primitive operations needed by such applications. In this paper we describe a highly configurable, Generic Multicast Transport Service (GTS), which supports the implementation of group-based applications in wide-area settings. GTS is unique in that it offers fault-tolerant, order-preserving multicast on arbitrary communication protocols, including e-mail. As another distinguishing mark, messages can be sent to processes even when they are temporarily unavailable, which permits disconnected operation and mobility. We further propose an object-oriented system design consisting of *adaptor objects* interconnected to form a *protocol tree*. Adaptor objects offer a common interface to dissimilar communication protocols, and make it easy to incorporate new protocols into GTS. Currently, GTS is being used in a cooperative software engineering environment and in other projects. GTS is available for anonymous ftp.

Keywords: Disconnected Operation, Replication, Message Spooling, Reliable Multicast, Wide Area Networks

1 Introduction

1.1 Motivation

Groupware, replicated file archives, and other kinds of distributed systems stimulate the need for structuring activities around process groups [3, 15] and reliable, order-preserving multicast [8]. We have developed a novel communication substrate, called the *Generic Multicast Transport Service (GTS)*, which enables the implementation of process group-based applications in wide-area networks. As the main abstractions, GTS offers reliable order-preserving multicast, reliable point-to-point communication, and process groups. A variety of transport protocols are supported and new protocols can be incorporated into the service easily. GTS is unique in that it permits disconnected operation, encrypted communication, reconfiguration, and in that applications may transmit messages without waiting until they have been delivered.

In the GTS system model, processes become unresponsive due to failures or due to disconnected operation of portable equipment. In such situations, GTS will spool messages on non-volatile storage and deliver them to their recipients as soon as they become available and register with GTS again. Addressing is by *Uniform Resource Locators* similar to the World Wide Web.

1.2 Related Work

Examples of state-of-the-art toolkits offering process groups and reliable, order-preserving multicast are Consul [12], Electra [10], Horus [14], Isis [4], and Transis [1]. These toolkits primarily aim to support applications running within one LAN. If a process has been unresponsive for a certain (usually short) period of time, their default behavior is to regard the process as faulty and to exclude it from the system. If a process or a whole group becomes unresponsive, applications cannot submit messages to it any more. Thus, periodic communication and disconnected operation are not adequately supported.

In contrast, GTS tolerates arbitrary communication delays and permits the sending of messages to disconnected processes, both by unicast[1] and multicast communication. In GTS, processes are never

*Research supported by grants from the Swiss National Science Foundation, Siemens-Nixdorf, Union Bank of Switzerland, and KWF/CERS

[1] point-to-point communication

excluded from the system unless explicitly requested by the user. Electronic retail banking, cooperative software engineering, software update protocols, distributed document servers, WAN applications to capture seismic signals, and replicated file archives are examples of applications which fit the GTS model. Moreover, GTS does not compete with the aforementioned toolkits but can be used in conjunction with them. Simply put, our scheme is well-suited for applications where disconnected operation, configurability, and widely distributed resources are more important than high-performance communication.

The ISIS wide-area facility [11] is the system most similar to GTS. At the heart of the facility lies a fault-tolerant application spooler which is restricted to a single LAN. ISIS applications can log messages into the spooler. An application that has failed will restart by initiating a spool replay operation, causing messages in the spooler to be played back to the application. During communication failures, messages are directed into a second spooler, called the *interLAN* area, and delivered to the destination after communication is re-established. The ISIS wide area facility supports process groups as well as totally ordered and causally ordered multicast. The main difference to GTS is that the ISIS wide-area facility is built on TCP, whereas GTS can run on virtually any communication protocol. Further, GTS embodies a flexible system design which permits to add functionality such as message compression or encryption easily. Last but not least, the wide-area facility is part of the ISIS toolkit whereas GTS is a stand-alone system. An important advantage of the ISIS facility is that it can be replicated over a LAN to increase availability. In contrast, a GTS LAN spooler is not replicated. However, availability can be increased by disk mirroring and by instantiating more than one GTS server process per LAN.

1.3 Organization of the Paper

The rest of the paper is structured as follows. Section 2 describes the system model, Uniform Resource Locators, and other important base concepts of GTS. GTS' system design is addressed in Section 3. In Section 4 the programming interface is presented along with simple example programs. Application experience with a cooperative software engineering environment based on GTS is reported in Section 5. Finally, Section 6 summarizes and concludes the paper.

2 Generic Multicast Transport Service

2.1 System Model

In our system model we distinguish between two kinds of processes: on one side are the GTS *servers*, which implement message spooling, reliable multicast, and unicast communication. On the other side stand the end-user *applications*, which use GTS. End-user applications can run on both mobile (laptops, palmtops, message pads) and immobile (hosts, workstations, PCs) equipment. A GTS server, along with the applications connected to it, makes up what we call a *cluster* (Figure 1). Typically, a cluster is contained in one LAN or within one mobile component. If an application in cluster A wants to send a message to an application in cluster B, it submits it to its server S_A, which in turn sends it to server S_B. Finally, S_B delivers the message to the end-user application.

GTS permits reliable unicast and multicast communication even when the underlying communication protocol is unreliable. In multicast communication, a sender application submits a message to a *group* of receiver applications. GTS guarantees that all members of the group receive the message, and that all members deliver messages in exactly the same order, which is called *totally ordered multicast* [8]. Multicast communication is useful for distributing the same data from one sender to a group of receivers efficiently, to perform computations redundantly, or to synchronize replicated data. GTS also supports groups of process groups. In WAN settings, this is useful for structuring large groups as a hierarchy of separately maintained subgroups.

A message to a disconnected destination is retained in the spooler of the GTS server which observes the disconnection. As was mentioned before, a mobile computer contains both a message spooler (i.e., a GTS server) and one or several applications. The server will try to deliver the message until it succeeds or until the destination is removed permanently from GTS. As a message travels through GTS, there will always be one copy of it in some spooler, and the server holding the copy is responsible for delivering it to the destination server or to the end-user application itself, if it runs in the server's cluster. If a GTS server fails, then nothing is lost since messages, group membership lists, and other important data are persistent.

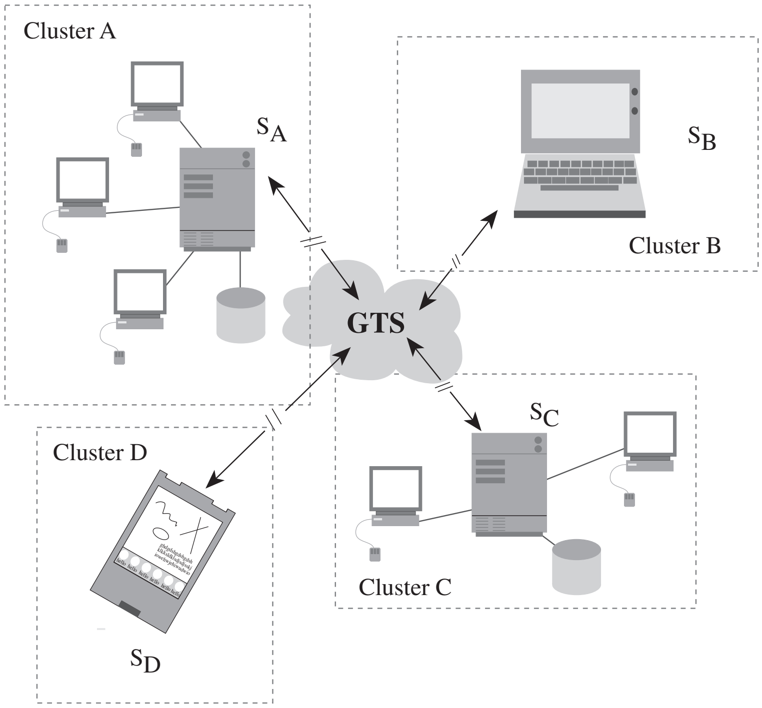

Figure 1: A typical GTS system configuration. S_i denotes the GTS server for cluster i. Applications are running on the workstations connected to their servers (Cluster A, C), on laptop computers (Cluster B), or on personal digital assistants (Cluster D).

2.2 Uniform Resource Locators

GTS supports an unrestricted set of protocols, for instance TCP, IP, AppleTalk, Mach Messages, ATM, e-mail, and UUCP. The API which programmers are confronted with is independent of the underlying transport protocols, and reliable multicast interprocess communication is feasible even with e-mail as the transport medium. In consequence, the addressing mechanism has to be simple and flexible. We decided to adopt the *Uniform Resource Locator (URL)* scheme proposed by the Internet Engineering Task Force. GTS URLs obey the following general form:

`protocol://cluster:server:localAddress/ticket`

The following GTS URLs all denote the same destination application:

```
tcp://ifi.unizh.ch:claude:9999/myApp
modem://ifi.unizh.ch:claude:(41)(1)3023570/myApp
uucp://ifi.unizh.ch:claude:Uclaude/myApp
email://ifi.unizh.ch:claude:gts/myApp
```

The first part of a URL defines the protocol used to deliver the message to its destination. The second part contains the address of the destination cluster. The `server` part contains the name of the GTS server. Hence, several GTS servers can run within the same cluster, if required. The `localAddress` is an internal, protocol-dependent address for the GTS server daemon, e.g. a TCP port number, a phone number, an e-mail account, and so forth. The `ticket` denotes the local application or application group the message is directed to. Here a character string stands for an endpoint application, a number for a group.

2.3 Privacy

GTS can be configured such that messages are transparently encrypted and decrypted using a public-key cryptosystem. Therefore, GTS maintains a public-key/private-key pair per URL. Typically, a GTS server provides the public keys of all destination URLs its client applications will send messages to, and it also maintains the private keys of its clients, such that incoming messages can be decrypted. If required, communication between a GTS server and its clients can be encrypted as well. Message encryption is accomplished by third-party software such as RSAREF or *Pretty Good Privacy (PGP)*. GTS' flexible system design (Section 3) permits easy incorporation of such encryption libraries.

2.4 Reliable Multicast Protocol

The multicast protocol we employ is similar to the one implemented in the AMOEBA [9] operating system. For each process group, there is one GTS server distinguished as the sequencer of the group. The sequencer maintains the URLs of the group members, and requests for joining or leaving the group must be directed to its sequencer.

To submit a multicast, the source server first delivers the message point-to-point to the sequencer server of the group (Figure 2). By inspecting the ticket of the destination URL, the server identifies the message as a multicast request, assigns the next multicast sequence number to it, looks up the URLs of the group members in a local membership file, and delivers the message to the member servers. Delivery is by one separate message per group member if the underlying transport protocol does not support multicast, or by one message for the whole group if all members can be reached with the same protocol, and given that the protocol supports multicast (e.g., IP with multicast extensions [2]). Note that a server can be sequencer and member of a group at the same time, and that member applications can be contained in the same cluster.

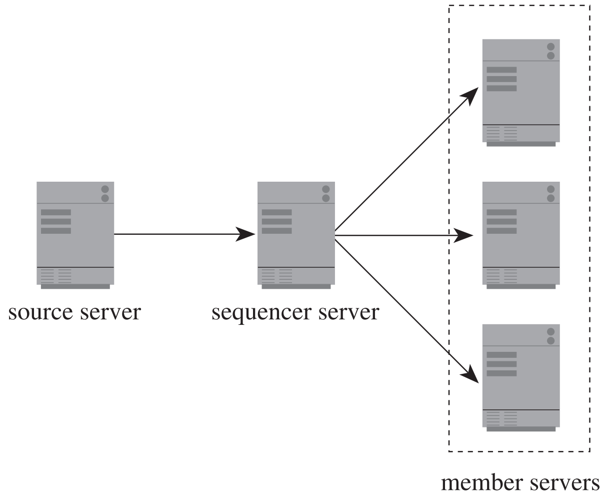

Figure 2: Message flow in the GTS multicast protocol.

Reliable multicast means that the members of a group agree on the set of multicasts they receive. If no damage to a source, sequencer, or member spooler occurs[2], GTS guarantees that a multicast is eventually received by all group members, and that members agree on the order of the multicasts they receive.

[2] e.g., due to a head-crash or a human lapse like accidentally deleting a spooler

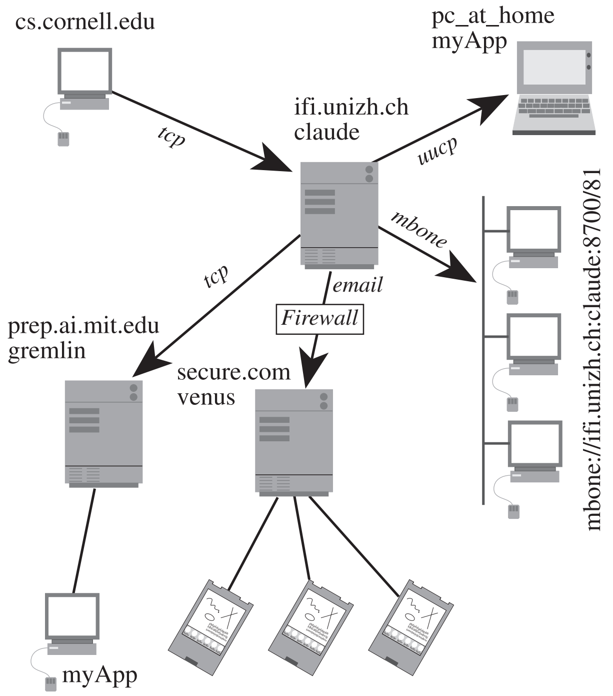

Figure 3: An example situation involving cascaded groups in a heterogeneous environment.

Consider the situation in Figure 3. Here, an application in the cluster `cs.cornell.edu` sends a message to the group
`tcp://ifi.unizh.ch:claude:9999/77`.
First, the sender's server delivers the message point-to-point to server `claude`. Since the ticket is a numeric value, `claude` retrieves the membership file for group 77 from its spooler. Assume that the file contains the entries
`tcp://prep.ai.mit.edu:gremlin:9999/myApp`,
`uucp://ifi.unizh.ch:claude:pc_at_home/myApp`,
`email://secure.com:venus:gts/31`, and
`mbone://ifi.unizh.ch:claude:8700/81`.
The message is multiplexed by server `claude` and forwarded to the four destinations. Since the third destination is a group maintained by the server of `secure.com`, which is reachable only by e-mail, the message is multiplexed by `venus` and transmitted to the message pads. The fourth destination also is a group; it contains members that can be reached by the IP multicast protocol [6] (also referred to as MBONE [2]).

3 Design of GTS

3.1 Protocol Tree

Several design goals guided the development of GTS:

- to support a wide range of protocols and operating systems,
- to make it easy for programmers to incorporate as yet unsupported protocols,
- to allow programmers to include their own API, and
- to devise a flexible design which other people can apply to their own systems.

This section focusses on the design of the GTS server, which is implemented in the C++ programming language. A GTS server is structured in a way similar to the *x*-kernel [13]. Each GTS server consists of a collection of *adaptor objects* plugged together to form a *protocol tree* as depicted in Figure 4. The root

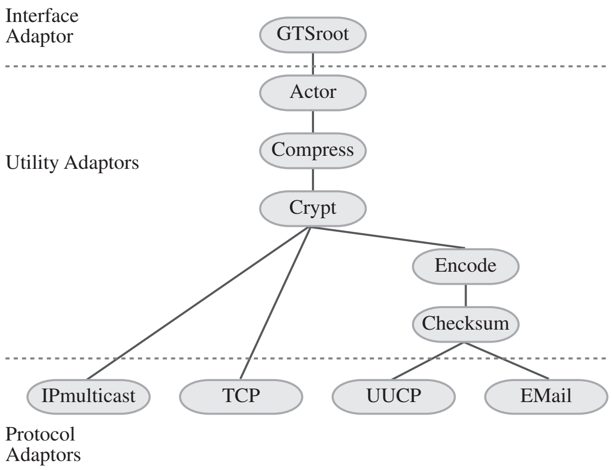

Figure 4: A sample protocol tree.

object (`GTSroot`) communicates with the client applications running in its cluster. Leaf objects, called protocol adaptors, perform unreliable message passing with remote servers by specific communication protocols. The utility adaptors in the middle area carry out tasks such as spooling and retransmission (the `Actor` object), compression, encryption, encoding, integrity check, and so forth in an operating system-independent fashion. Each adaptor object passes the messages it receives down the tree to one of its child adaptors. A message is routed through the tree according to the protocol part of its destination URL until it reaches a protocol adaptor. Finally, the protocol adaptor transmits the message to the destination server by the protocol it encapsulates.

At the destination server, the message is received by a protocol adaptor and is passed up the tree. If needed, it is checked, decoded, decrypted, and decompressed by the utility adaptors. The `Actor` adaptor spools the message to permit retransmission in case the end-user application is not available. Finally, the received message arrives at the `GTSroot` object where it is transmitted to the end-user application. Adaptor objects are organized in the form of the inheritance hierarchy depicted in Figure 5. Owing to this flexible system design, more than 90% of GTS' program code could be realized in a protocol- and operating system-independent way.

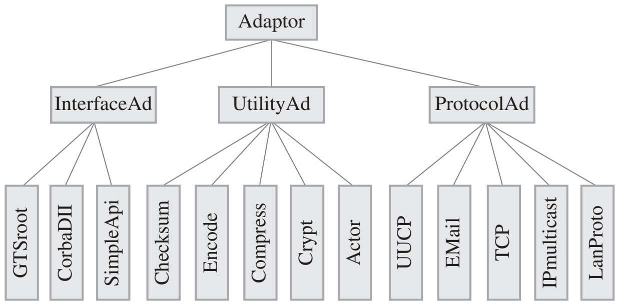

Figure 5: Adaptor inheritance hierarchy.

3.2 Adaptor Interface

A GTS adaptor object obeys the following interface:

```
class Adaptor {
public:
    virtual boolean down(Message&);
    virtual boolean up(Message&);
    virtual boolean done(Message&);
    virtual boolean viewChange(Message&);
    virtual boolean flush();
    virtual Adaptor& attach(Adaptor&);
};
```

An adaptor's `down` method is invoked by its father adaptor to pass down a message. To pass up a message, an adaptor invokes its father's `up` method. If an adaptor wants to discard a message (because it is corrupted or because it was received by the end-user application) it invokes the `done` method of all its child adaptors to ensure that they discard copies of the message they might hold. To force an adaptor to pass down the messages it stores, its `flush` method is invoked. The `viewChange` method informs an adaptor that an application joined or left a local group. Finally, the `attach` method is used to attach child adaptors and thus to construct a protocol tree.

For example, consider an adaptor to compress messages. Its `down` method compresses the data in the message, whereas its `up` method uncompresses the data. `done`, `viewChange`, `flush`, and `attach` need not be overwritten, thus the default behavior implemented in class `Adaptor` is inherited.

4 Using GTS

4.1 Programming Interface

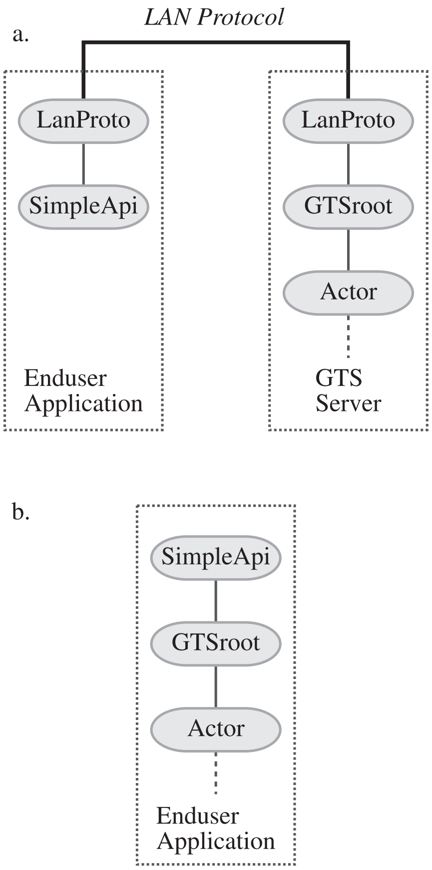

Figure 6: Coupling end-user applications with their GTS server.

In a LAN, end-user applications are linked with a communication stub that governs interaction with the cluster's server (with the `GTSroot` object, more exactly). This stub consists of an interface adaptor (`SimpleApi`) connected to a protocol adaptor (Figure 6 a.). The interface adaptor serves as an API to the programmer, whereas the protocol adaptor is used to communicate with the GTS server by a reliable LAN protocol like TCP or AppleTalk. In case GTS is installed on a mobile computer, end-user applications use the same interface adaptor as above, but can be directly linked with the server if required (Figure 6 b.). The present version of GTS provides the API below. As future work we plan to implement an API compatible with the CORBA *Dynamic Invocation Interface* [7].

```
class SimpleApi: public InterfaceAd {
public:
    // non-blocking send:
    boolean send(URL destination, const Message&);

    // blocking receive:
    boolean receive(OUT Message&,
        URL source =anybody);
    // non-blocking receive (polling):
    boolean receive(OUT Message&,
        OUT boolean& dataReady,
        URL source =anybody);

    // create a local URL group:
    boolean groupCreate(OUT unsigned int& groupID);
    // destroy a local URL group:
    boolean groupDestroy(unsigned int groupID);
    // join a URL group:
    boolean groupJoin(URL group, URL member);
    // leave a URL group:
    boolean groupLeave(URL group, URL member);
    // obtain the members of a local group:
    boolean groupGetInfo(URL group,
        OUT List& members);

    // obtain the URL of an application:
    boolean getMyURL(OUT URL& me);
};
```

The `send` operation submits a message to a destination URL without blocking the sender. To be suspended until a message has arrived, applications issue the first version of `receive`. To wait for a message from a specific sender, `source` is set to the URL of that sender. Otherwise the first arriving message is returned, regardless of the sender. The second version of `receive` is used to check whether a message is available without being blocked. If a message is available, it is assigned to the `Message` parameter and `dataReady` is set to `TRUE`. `groupCreate`/`groupDestroy` are used to create/destroy a group on the local server, `groupID` holds the group ID. `groupJoin` is used to add a member to a group, `groupLeave` to remove a member. `groupGetInfo` returns a list containing the URLs of the members of `group`. Finally, `getMyURL` serves to obtain the URL of an application.

4.2 Configuration

Various parameters of the GTS server can be tailored in a configuration file. For instance, permission to join or leave a group may be explicitly granted only to applications running in a certain cluster:

```
# ACCESS RIGHTS FOR LOCAL GROUPS:
#
# everybody can join or leave group 1:
group 1     jl     *
# only applications in cluster foo.edu can join group 2:
group 2     j      foo.edu
```

The return address assigned to an outgoing message can be controlled as well. This is useful for tailoring a GTS installation to environments where a firewall allows TCP traffic only in one direction, for instance.

```
# RULES TO SET THE RETURN ADDRESSES OF
# OUTGOING MESSAGES:
#
# The filters below are applied in sequence to the
# destination URL of an outgoing message until a
# match occurs.
#
# We request that replies to messages sent to bitnet
# or smallfirm.com destinations be returned to us by
# email:
myurl   *.bitnet         email
myurl   smallfirm.com    email
# uucp destinations shall submit replies by uucp:
myurl   *.uucp           uucp
# for the rest of the world we choose TCP:
myurl   *                tcp
```

For a cluster called `secure.com` residing behind a firewall (Figure 3), the `myurl` entries could be set as follows:

```
myurl    *secure.com    tcp
myurl    *              email
```

This means that within the `secure.com` domain, messages are exchanged by TCP. Replies for messages delivered to applications outside the domain will automatically arrive by e-mail to bypass the firewall. Thus, messages will be delivered by TCP in one direction and by e-mail in the other direction. For security, GTS can be set up such that messages crossing domain boundaries are protected by public key encryption (Section 2.3).

Protocol adaptors are configured as follows:

```
# CONFIGURATION OF PROTOCOL ADAPTORS:
#
protocol tcp    200000 30   300   9999 unicast
protocol email  100000 3600 28800 gts  unicast
protocol mbone  2048   30   300   9999 multicast tcp
```

In the above protocol configuration, the second entry assigns a name to the protocol adaptor. Messages traveling through the adaptor are split into fragments whose maximum size is controlled with the third entry. The fourth entry gives the initial retransmission interval in seconds, whereas the fifth entry the maximum interval. A retransmission interval is constantly increased until it reaches the maximum value. The sixth entry defines an address the adaptor uses to check for messages, for instance a TCP port number or an e-mail account. The seventh entry declares whether the underlying protocol supports multicast. In case of a multicast protocol, a further entry is supplied to specify by which protocol failed messages are retransmitted. For instance, when a multicast is sent through the `mbone` adaptor it is transmitted to the group by IP multicast [6, 2]. If the message fails to arrive at some destinations, it is transmitted point-to-point to them by `tcp`. If an adaptor does not support multicast, the `Actor` adaptor (Section 3.1) transparently multiplexes a multicast to one message per group member.

4.3 Examples

In the following example, a client application sends a request message to a weathermap server application, and then is suspended until a reply has arrived:

```
// client application:
//
SimpleApi gts;
...
Message request, reply;
request << "send weathermap of Detroit";

// submit the request:
gts.send("tcp://arc.nasa.gov:explorer:9999/mapServer",
    request);

// wait for the reply from the map server:
gts.receive(reply,
    "tcp://arc.nasa.gov:explorer:9999/mapServer");

// process received data ...
```

```
// server application:
//
SimpleApi gts;
...
Message request, reply;
// wait for a request message from any source:
gts.receive(request);

// process the request ...
// return a reply to the sender:
gts.send(request.getTrueFrom(), reply);
```

If the request is to be transmitted by e-mail, it is sufficient to change the above destination URL to `email://arc.nasa.gov:explorer:gts/mapServer`

In the next example, an application creates a group on its local GTS server and sends a multicast to it:

```
SimpleApi gts;
...
// variable to hold the group ID:
unsigned int gid;

// get the URL of this application:
URL group;
gts.getMyURL(group);

// create a group on the local server.
// Its group ID is assigned to gid:
gts.groupCreate(gid);

// modify the ticket of my URL to obtain
// the URL of the group:
group.setTicket(gid);

Message msg;
msg << "Hello World";

// join members to the group:
gts.groupJoin(group,
    "tcp://prep.ai.mit.edu:gremlin:9999/myApp");
gts.groupJoin(group,
    "uucp://ifi.unizh.ch:claude:pc_at_home/myApp");
gts.groupJoin(group,
    "email://secure.com:venus:gts/31");
gts.groupJoin(group,
    "mbone://ifi.unizh.ch:claude:8700/81");

// submit a multicast:
gts.send(group, msg);
```

5 Application Experience

At the UBILAB we are currently developing Beyond-Sniff, a platform with tools to support cooperative software engineering [5]. For developers connected by networks with high communication bandwidth (i.e. in LANs), cooperation is made possible by a number of distributed infrastructure services which rely on Beyond-Sniff's own mechanisms for data and control integration. Cooperation support over networks with low bandwidth and with portable destinations which are only temporarily active is based on a replication mechanism.

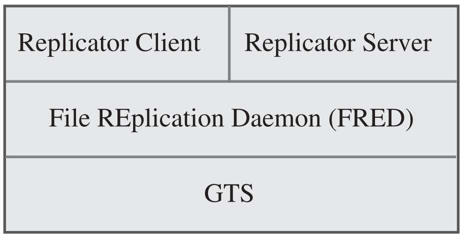

Figure 7: Architecture of the Beyond-Sniff Replicator.

In the mid-term we intend to integrate the replication mechanism into Beyond-Sniff's infrastructure services such that it will become as transparent as possible to developers. To gain hands-on experience on how a replication-based approach influences software development in widely distributed teams, we have implemented and successfully applied the Replicator, a stand-alone replication mechanism based on GTS. The purpose of the Replicator is to reliably synchronize any number of remote client directories with a master directory tree. Its architecture consists of three levels as depicted in Figure 7.

The lowest level is formed by GTS which guarantees reliable data transfer between the master and the client sites over various protocols and in spite of temporarily disconnected clients. The GTS File Replication Daemon (Fred) creates, packages, transmits, and applies incremental updates to a replicated directory tree. Fred provides a simple command-line interface and can be used either as a background daemon, synchronizing automatically at well-defined time intervals, or as a slave synchronizing on demand. The highest level is formed by the Replicator's graphical user interface which consists of a server (Figure 8) and a client part (Figure 9). The interface mainly serves to let users easily access Fred's functionality.

From the user's perspective, the Replicator works as follows. To set up a synchronization group, the administrator of the master copy defines the directory to be mirrored and a further directory needed to calculate incremental updates. The administrator then defines which clients will be supported by specifying their URLs as well as the personal e-mail addresses of the persons in charge of administering the clients. When a new client joins a replication group, a complete version of the directory hierarchy is packaged and sent to it. When a directory is updated, the packaging and multicasting of updates to the clients is triggered explicitly on the master site.

The client administrators are automatically notified by an e-mail message when a data transfer took place. The graphical user interface on the client side (Figure 9) mainly presents lists with new and already applied updates. Updates are explicitly applied when it makes sense in the context of the ongoing cooperation.

The implementation of Fred and of the Replicator's user interface was straightforward (about 10 working days) and it has proven the usefulness of GTS as well as the adequacy of its API. Without GTS we would not have been able to develop the Replicator in a reasonable time. The Replicator is successfully being applied to synchronize our joint development efforts between sites in Switzerland, Germany, and Austria. We intend to make the Replicator publicly available in the near future.

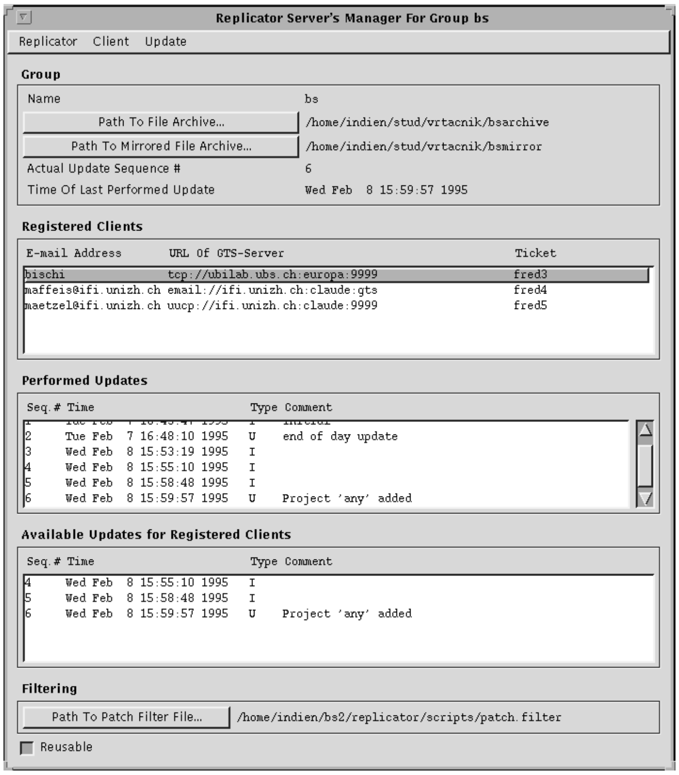

Figure 8: User interface of the Replicator server.

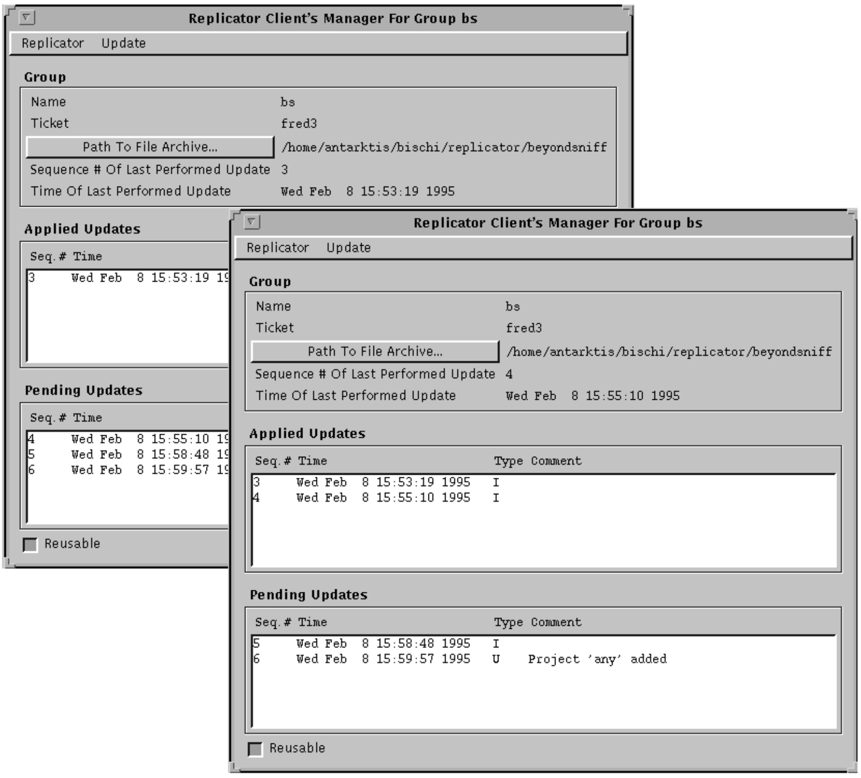

Figure 9: User interface of the Replicator client.

6 Conclusions

Widely-distributed systems often need to synchronize replicated data in spite of disconnected equipment and failures. In this paper we presented a novel communication substrate, called the *Generic Multicast Transport Service (GTS)*, which was developed at the University of Zurich and at the Union Bank of Switzerland. The development of GTS was influenced by the results of projects such as AMOEBA, ELECTRA, ISIS, and x-kernel. GTS is different from previous work on process group-based systems mainly in that it deals with disconnected operation and in that it focusses on widely-distributed rather than on local resources. Moreover, a flexible, object-oriented system design consisting of *adaptor objects* interconnected to a *protocol tree* has been devised. This system design permits reliable multicasts to be issued on arbitrary transport protocols, for example on TCP/IP or even on e-mail, if necessary, and messages are addressed to Uniform Resource Locators. New functionality can be added to GTS easily by developing new plug-in adaptors.

Presently, GTS is being used to build heterogeneous distributed applications interconnecting several clusters. As an example of a real-world application employing GTS we described Beyond-Sniff, a cooperative software engineering environment. In our experience, GTS is ideal for replicating data in a distributed system consisting of static and mobile computing equipment. We also found that groupware serving asynchronous forms of collaboration often requires the kind of system support this paper proposes.

As future work we plan to port GTS to PC operating systems and to personal digital assistants. We also intend to incorporate GTS into a CORBA event channel service such that widely distributed CORBA objects can communicate through GTS.

Availability

GTS is available for anonymous ftp in the directory `ftp://ftp.ifi.unizh.ch/pub/projects/gts/`. Information on Beyond-Sniff and on the Replicator can be retrieved from `ftp://ftp.ubilab.ubs.ch`. Universities can obtain a free copy of the non-distributed SNiFF+ programming environment from `ftp://self.stanford.edu/pub/sniff/`.

References

[1] AMIR, Y., DOLEV, D., KRAMER, S., AND MALKI, D. Transis: A Communication Sub-System for High Availability. In *22nd International Symposium on Fault-Tolerant Computing* (July 1992), IEEE.

[2] BAKER, S. Multicasting for Sound and Video. *Unix Review* (Feb. 1994).

[3] BIRMAN, K. P. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM 36*, 12 (Dec. 1993).

[4] BIRMAN, K. P., AND VAN RENESSE, R., Eds. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[5] BISCHOFBERGER, W. R., KOFLER, T., MÄTZEL, K.-U., AND SCHÄFFER, B. Computer Supported Cooperative Software Engineering with Beyond-Sniff. In *Proceedings of the 7th Conference on Software Engineering Environments* (Noorwijkerhout, The Netherlands, 1995).

[6] DEERING, S. Host Extensions for IP Multicasting. RFC 1112, Request for Comments, Aug. 1989.

[7] DIGITAL EQUIPMENT CORP., HEWLETT-PACKARD CO., HYPERDESK CORP., NCR CORP., OBJECT DESIGN INC., SUNSOFT INC. *The Common Object Request Broker: Architecture and Specification*, Dec. 1993. Revision 1.2.

[8] HADZILACOS, V., AND TOUEG, S. Fault-Tolerant Broadcasts and Related Problems. In *Distributed Systems*, S. Mullender, Ed., second ed. Addison Wesley, 1993, ch. 5.

[9] KAASHOEK, M. F., TANENBAUM, A. S., HUMMEL, S. F., AND BAL, H. E. An Efficient Reliable Broadcast Protocol. *ACM SIGOPS Operating Systems Review 23*, 4 (Oct. 1989).

[10] MAFFEIS, S. A Flexible System Design to Support Object-Groups and Object-Oriented Distributed Programming. In *Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming* (1994), R. Guerraoui, O. Nierstrasz, M. Riveill, Ed., Lecture Notes in Computer Science 791, Springer-Verlag.

[11] MAKPANGOU, M., AND BIRMAN, K. Designing Application Software in Wide Area Network Settings. Tech. Rep. 90-1165, Department of Computer Science, Cornell University, Oct. 1990.

[12] MISHRA, S., PETERSON, L. L., AND SCHLICHTING, R. D. Consul: A Communication Substrate

for Fault-Tolerant Distributed Programs. *Distributed Systems Engineering Journal 1*, 2 (Dec. 1993).

[13] PETERSON, L., HUTCHINSON, N., O'MALLEY, S., AND RAO, H. The x-kernel: A Platform for Accessing Internet Resources. *IEEE Computer 23*, 5 (May 1990).

[14] VAN RENESSE, R., AND BIRMAN, K. P. Fault-Tolerant Programming using Process Groups. In *Distributed Open Systems*, F. Brazier and D. Johansen, Eds. IEEE Computer Society Press, 1994.

[15] VERÍSSIMO, P., AND RODRIGUES, L. Group Orientation: A Paradigm for Distributed Systems of the Nineties. In *Proceedings of the Third Workshop on Future Trends of Distributed Computing Systems* (Apr. 1992), IEEE Computer Society.