GTS: A Generic Multicast Transport Service

Silvano Maffeis, Walter Bischofberger, and Kai-Uwe Mätzel

Information Technology Laboratory (UBILAB)
Union Bank of Switzerland
and
University of Zurich
Department of Computer Science

UBILAB Technical Report 94.6.1

June 1994

Abstract

Many nation-wide distributed applications can profit from process-groups and reliable multicast communication, 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. The GTS is unique in that it makes possible reliable, 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 is achieved by making messages persistent. 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 the GTS. Currently, the GTS is being used in a cooperative software engineering environment and in other distributed applications. The GTS is available for anonymous ftp.

1 Introduction

1.1 Motivation

Groupware, computer supported cooperative work, 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, fault-tolerant, heterogeneous applications on wide-area networks. As the main abstraction, the GTS provides reliable order-preserving multicast, reliable point-to-point communication (unicast), and process-groups. A variety of transport protocols are supported and new protocols can be incorporated into the service easily. The GTS is unique in its flexibility, in that it does not require that communicating processes be available at the same time, and in that it allows temporarily unavailable processes to remain in their process-groups. To that purpose, the GTS will spool messages on non-volatile storage and deliver them to their recipients as soon as they become available and register with the GTS again.

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]. The default policy of these toolkits is to remove failed members from their groups. A recovered process will have to re-join the system, and to request the current application state from a group member. If all members of a group fail, processes which depend on the group cannot make progress any more. This scheme is well-suited for applications requiring minimal communication delays and where interacting processes are available at the same time.

In contrast, the GTS aims at supporting asynchronous applications which tolerate long communication delays, and where the send operations of a process must progress even when recipient processes are unavailable owing to a planned off-time or failure. Cooperative software engineering applications, software update protocols, multimedia messaging systems, distributed document servers, and replicated file archives are examples of such applications. The GTS is tailored for applications spawning several LANs, though applications running within the boundaries of one LAN can be accommodated as well. Moreover, the GTS does not compete with the forementioned toolkits, but can be used in conjunction with them. Simply put, our scheme is well-suited for applications where configurability, heterogeneity, and persistent messages are more important than minimal communication latency.

2 The Generic Multicast Transport Service

2.1 System Model

In our system model we distinguish 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 enduser *applications*, which use the GTS. A GTS server, along with the applications connected to it, makes up what we call a *cluster* (Figure 1). A cluster is contained in one LAN. If an application in cluster A wants to send a message to an application in cluster B, it submits it to server S_A, which in turn sends it to server S_B. Finally, S_B delivers the message to the destination application.

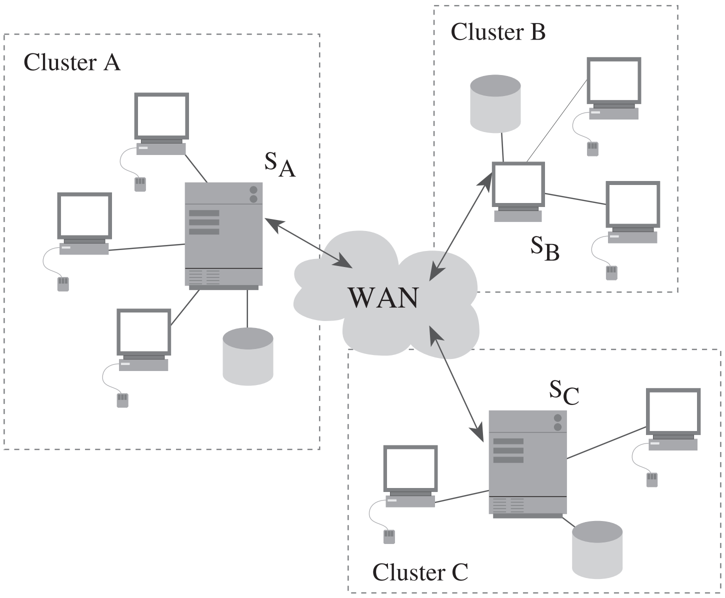

Figure 1: A typical GTS system configuration. S_i denotes the GTS server for cluster i. Applications are running on the workstations attached to the servers.

The GTS offers reliable unicast and multicast communication, even when the underlying communication protocols are unreliable. In multicast communication, a sender application submits a message to a *group* of receiver applications. The GTS guarantees that all members of the group receive the message, and that all receivers deliver all messages in exactly the same order, which is called *totally ordered multicast*. In contrast to most existing toolkits, the GTS also supports groups of process-groups. In WAN settings, this is useful for structuring large groups as a hierarchy of separately maintained sub-groups.

Applications may become unavailable as a result of hardware faults, software errors, planned off-times, client mobility, or human lapses. A message to an unreachable destination is retained in the spooler of the

GTS server which observes the fault[1]. The server will try to deliver the message until it succeeds or until the destination is detached permanently from the GTS by a system administrator. As a message travels through the 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 destination 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.

2.2 URL Based Addressing

The GTS supports an unrestricted set of protocols, for instance TCP, IP, AppleTalk, Mach Messages, ATM, and even e-mail. 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 transport medium. In consequence, the addressing scheme employed has to be simple and flexible. We decided to adopt the *Uniform Resource Locators (URL)* proposed by the Internet Engineering Task Force. GTS URLs obey the following general form:

`GTSprotocol://GTSserver:localAddress/memberID`

for example:

`gts-tcp://claude.ifi.unizh.ch:9999/77`, or

`gts-email://ifi.unizh.ch:gts/77`.

The first part of a URL denotes the protocol a GTS server will use to deliver messages to the destination server. The second part contains the address of the destination server in a protocol-dependent notation. The `localAddress` is an internal address, e.g. a port number, a directory, or an e-mail account. Finally, the `memberID` is an integer value which denotes the application process or process-group the message is directed to.

2.3 Reliable Multicast Protocol

The multicast protocol employed is similar to the one implemented in the AMOEBA [9] operating system. To guarantee total ordering and reliable delivery, a GTS server acts as *sequencer* for the groups it maintains. To submit a multicast, an application point-to-point delivers the message to its GTS server. By inspecting the `memberID` 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 list, and reliably delivers the message. Delivery is by one separate unicast message per group member if the transport protocol does not support multicast, or by one message for the whole group if all members can be reached by the same protocol, and given that the protocol supports multicast (e.g., IP with multicast extensions [2]).

2.4 Persistent Data Facilities

When an application wants to send a message, it always submits it to the GTS server in its cluster. The server will first spool the message to avoid losing it in case of a failure. Thereafter, the message is delivered to the destination server by the transport protocol specified in the destination URL. As soon as the remote server has acknowledged the receipt, the local server can delete the message from its spooler. If the remote server is unavailable, the message is retransmitted periodically until the server acknowledges the receipt.

Thus, the GTS server represents a central point of failure. This problem is alleviated by the fact that relevant data are kept on non-volatile storage and that, once recovered, a GTS server can carry on with its work. Moreover, one or several backup servers can be configured per cluster and applications will automatically switch to one of them when the primary GTS server becomes unreachable. Membership lists for the groups maintained by a server are also persistent. A group membership list contains the URLs of the destinations which make up the group, and the members of a group need not support the same protocol.

[1] by a timeout mechanism or by a hint from the operating system.

3 Design of the GTS

Several design goals guided the development of the GTS: (a) to support a wide range of protocols and operating systems, (b) to make it easy for programmers to incorporate as yet unsupported protocols, (c) to make it easy to add functionality such as message compression and public-key encryption, (d) to allow programmers to include their own API, and (e) 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] and to Electra [11]. Each GTS server consists of a collection of *adaptor objects* plugged together to form a *protocol tree* as depicted in Figure 2. The root object (`GTSroot`) communicates with the client applications running in its cluster. Leaf

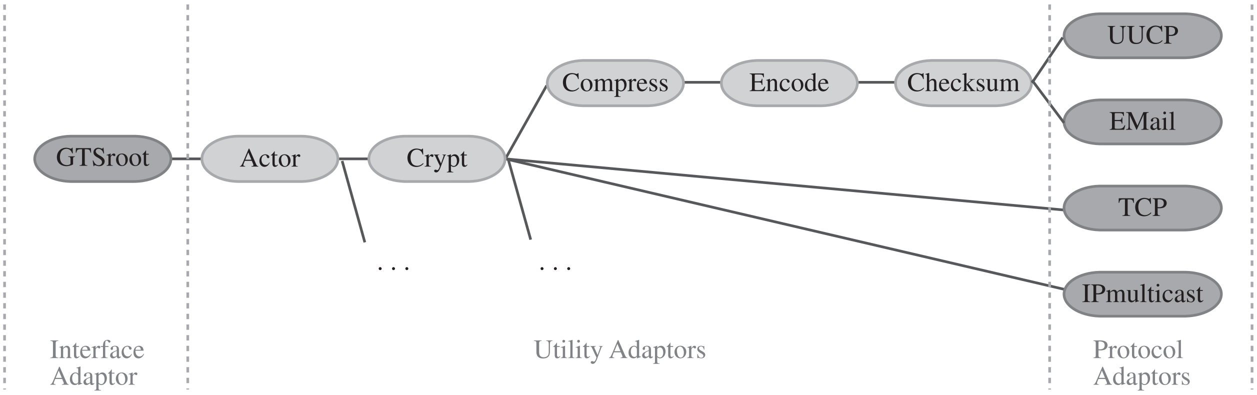

Figure 2: A sample protocol tree.

objects, called protocol adaptors, perform unreliable message passing on specific communication protocols. The utility adaptors in the middle area carry out tasks such as reliable communication (the `Actor` object), encryption, compression, encoding, integrity check, and so forth in a generic 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 its destination URL until it reaches a protocol adaptor. Finally, the protocol adaptor transmits the message by the protocol it encapsulates.

At the destination GTS server, the message is received by a protocol adaptor and is passed up the tree. If needed, it is checked, decoded, decompressed, decrypted, and assembled by the utility adaptors. Finally, the received message arrives at the `GTSroot` and is transmitted to the destination application in the cluster.

Adaptor objects are coupled abstractly to simplify the task of configuring a protocol tree. Adaptors are interconnected by the methods each concrete adaptor class inherits from the `Adaptor` abstract base class. The resulting class hierarchy is shown in Figure 3. Owing to this flexible system design, more than 90% of the GTS' program code could be realized in a protocol and operating system independent way.

4 Programming Interfaces

Enduser applications are linked with a communication stub that governs the interaction with the local server. This stub consists of an interface adaptor connected to a protocol adaptor. The interface adaptor serves as API to the programmer, whereas the protocol adaptor is used to communicate with the GTS server (with the `GTSroot` object, more exactly) by a reliable LAN protocol. Actually, two different APIs are provided: `SimpleApi`, which offers the interface below and `CorbaDII`, which offers an interface compatible with the *CORBA Dynamic Invocation Interface* [7]. The `CorbaDII` class differs from the `SimpleApi` class in that it specifies operations to marshal parameters and to send a message to an object representing an abstract service.

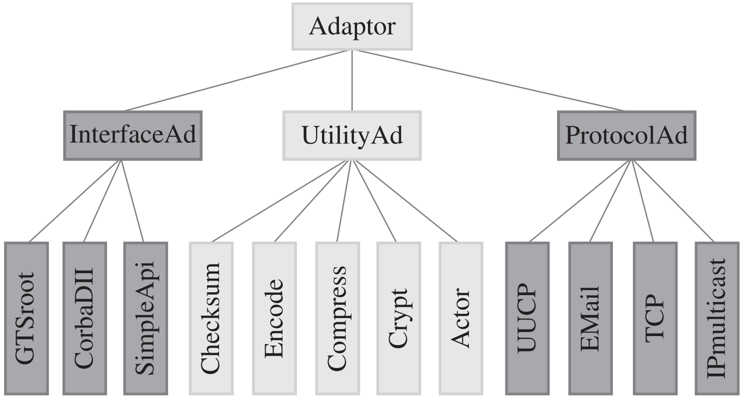

Figure 3: Adaptor inheritance hierarchy.

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

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

    // create a group on the local server:
    boolean groupCreate(OUT unsigned int& groupID);
    // destroy a local group:
    boolean groupDestroy(unsigned int groupID);
    // obtain the members of a local group:
    boolean groupGetInfo(unsigned int groupID, OUT List<URL>& members);
    // join a local or remote group:
    boolean groupJoin(URL server, unsigned int groupID, URL newMember);
    // leave a local or remote group:
    boolean groupLeave(URL server, unsigned int groupID, URL newMember);
};
```

5 Application Experiences

In a joint effort, the Union Bank of Switzerland, Siemens-Nixdorf, and the University of Zurich are developing BEYOND-SNIFF, a novel platform to support cooperative software engineering environments. BEYOND-SNIFF employs the GTS as communication substrate for information propagation and data replication over wide-area connections. BEYOND-SNIFF provides a proper infrastructure for distributed, cooperative work. It offers tools and frameworks for implementing service based architectures, in which services and large amounts of data can be shared. On top of the BEYOND-SNIFF platform a cooperative software engineering environment has been built. New tools focussing on cooperation were implemented and SNiFF+ [5, 6], a full-fledged C/C++ programming environment has been ported to BEYOND-SNIFF.

Development of the GTS was initiated in this context to elaborate the communication primitives required for supporting distributed workgroups in their cooperation on the same software engineering projects. The GTS is used to deliver the project relevant information to project members residing in different LANs by reliable and totally ordered multicast. The GTS has several indispensable properties which are necessary to support cooperation between distributed workgroups within BEYOND-SNIFF: (a) Reliable unicast and totally ordered multicast that works even when members of a group are not available at the time information is being distributed. (b) Support for disconnected operation and thus for client mobility. (c) Support for different communication protocols including e-mail as the "smallest common denominator" protocol. (d) A flexible adaptor architecture to fulfill different security requirements.

6 Conclusions

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 GTS is influenced by results of projects such as AMOEBA, ELECTRA, ISIS, and *x*-kernel, and enables the implementation of fault-tolerant distributed applications in wide-area settings. It is different from previous work on process-group based systems in that it focusses on widely distributed rather than on local resources, and in that groups of applications can interact even when some applications are not available at the time information is being distributed. Moreover, a flexible, object-oriented system design consisting of *adaptor objects* interconnected to a *protocol tree* has been devised. This system design permits to issue reliable multicasts on arbitrary transport protocols such as TCP/IP or even e-mail. New transport protocols can be incorporated easily by developing adaptors for them.

At the Union Bank of Switzerland and at the University of Zurich the GTS is being used to build heterogeneous distributed applications interconnecting several clusters, each containing UNIX systems and PCs. We run group based applications even with e-mail as transport protocol, since it often is the only means for communicating with mainframes or PCs. As an example of an application employing the GTS we described BEYOND-SNIFF, a platform to support cooperative software engineering environments. The GTS is particularly well-suited for asynchronous, group-based applications which tolerate communication delays and repair times. In our experience, groupware serving asynchronous forms of collaboration requires the kind of system support this paper proposes.

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. Sniff – A Pragmatic Approach to a C++ Programming Environment. In *USENIX C++ Conference* (Portland, Aug. 1992), The USENIX Association.

[6] Bischofberger, W. R., Kofler, T., and Schaeffer, B. Object-Oriented Programming Environments: Requirements and Approaches. *Software — Concepts and Tools, Springer-Verlag 15*, 2 (1994).

[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. 1991. Revision 1.1, OMG Document Number 91.12.1.

[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. Electra – Making Distributed Programs Object-Oriented. In *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems IV* (San Diego, CA, 1993), USENIX.

[11] 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.

[12] Mishra, S., Peterson, L. L., and Schlichting, R. D. Consul: A Communication Substrate for Fault-Tolerant Distributed Programs. Tech. rep., Department of Computer Science, The University of Arizona, 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.