The Interactive Sharing Transfer Protocol Version 1.0

Richard C. Waters, David B. Anderson, Derek L. Schwenke

TR97-10   October 1997

Abstract

The Interactive Sharing Transfer Protocol (ISTP) supports the sharing of information about a virtual world among a group of user processes. ISTP allows the interoperation of diverse systems running on diverse hardware. The key advantages of ISTP are that it supports (1) real-time interaction between users, (2) the communication of every kind of information required in a virtual world, and (3) scalability to large numbers of simultaneous users and large virtual worlds.

'Design of the Interactive Sharing Transfer Protocol', WET ICE '97 – IEEE Sixth Workshops on Enabling Technologies for Collaborative Enterprises: Infrastructure for Collaborative Enterprises, IEEE Computer Society Press, Los Alamitos, CA, June 1997, pp. 140-147.

MERL – A MITSUBISHI ELECTRIC RESEARCH LABORATORY
http://www.merl.com

The Interactive Sharing Transfer Protocol Version 1.0

Richard C. Waters
David B. Anderson
Derek L. Schwenke

TR-97-10 October, 1997

Abstract

The Interactive Sharing Transfer Protocol (ISTP) supports the sharing of information about a virtual world among a group of user processes. ISTP allows the interoperation of diverse systems running on diverse hardware. The key advantages of ISTP are that it supports (1) real-time interaction between users, (2) the communication of every kind of information required in a virtual world, and (3) scalability to large numbers of simultaneous users and large virtual worlds.

(Part I of this report is based on Waters R.C., Anderson D.B., & Schwenke D.L., "Design of the Interactive Sharing Transfer Protocol", *Postproc. WET ICE '97 – IEEE Sixth Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, (MIT, June 1997), IEEE Computer Society Press, Los Alamitos CA, 1997.)

Contents

Part I
Overview

This report describes the Interactive Sharing Transfer Protocol (ISTP) in full detail. This first part of the report stands by itself as a high level overview of the protocol.

The second part describes each individual message in the protocol. It is a complete specification of what each bit in each message means, but is lacking in motivation and discussion of how the various messages interact.

The third part of the report describes the five subprotocols that comprise ISTP. It explains how the various ISTP messages interact to support the subprotocols and discusses why the subprotocols are designed the way they are.

Parts II and III are best read together, with Part II used as a reference when the various messages are discussed in Part III.

ISTP is used for communication by the Scalable Platform for Large Interactive Networked Environments (Spline). Spline, which is described in separate documents [11, 12], makes it easy to build Distributed Virtual Environments (DVEs)—virtual worlds where multiple people interact with each other and computer simulations in a 3D visual and audio environment [10]. Using ISTP, Spline performs all the processing necessary to maintain a distributed, modifiable, and extendable model of a virtual world that is shared between the participants.

It is important to realize that while ISTP and Spline have evolved together, they are completely separate. The Spline API could be supported by other communication protocols and ISTP could be used to support other APIs.

1 Goals of ISTP

The communication needs of Distributed Virtual Environments (DVEs)—virtual worlds where multiple people interact with each other and computer simulations in a 3D visual and audio environment—are served best by a transport layer that simultaneously satisfies the following goals.

Near real-time – State changes communicated as fast as possible–i.e. in fractions of a second.
Near state equality – Processes closely agree on the state of the data they share.
Scalability – Tens of thousands of simultaneous users and hundreds of thousands objects supported.
Full malleability – Communication of all DVE data, so applications can be arbitrarily extended at run time.
WWW-like – Very flexible interaction of authors and viewers, so any user can contribute content.
Low bandwidth – Operation on PCs connected to the current Internet by modems.

The first goal is qualified because current wide area network realities make true real-time interaction impossible. The most you can ask of a DVE transport layer is that it use the fastest communication protocols available.

The second goal is qualified because it is impossible to simultaneously provide near real-time

	CS	DIS	ISTP
Near real-time	B	A	A
Near state equality	A	A-	A
Scalability		C	A-
Full malleability	C		A
WWW-like			A
Low bandwidth	A	C	A

Figure 1: Rating communication approaches.

interaction, exact equality of dynamic shared data, and scalability. The handshaking required to insure exact equality between more than a few shared copies adds latency that makes real-time interaction impossible. To achieve fast interaction, one must allow temporary disagreements between processes about the state of shared data. Fortunately, DVEs (unlike airline reservation systems) can typically be constructed to tolerate such disagreements.

Malleability is important because, unlike single-user VR, many-user DVEs are inherently non-stop in nature. Even if no single user is connected for more than a short time, long periods of time are likely to pass before the total number of users ever drops to zero. In this kind of scenario, rapid evolution of a DVE is practical only if it can be done while the system is running.

WWW-like operation is important so that the horizon of DVEs can be rapidly expanded by the addition of content from many sources. The creation of new content is too important to be relegated only to centralized organizations. It must be easy for an arbitrary user to field new content that is integrated with preexisting content.

To support malleability and WWW-like operation, a DVE transport layer must support the communication of such things as graphical models, recorded sounds, behaviors, and descriptions of new object classes. Because such data items are large, they cannot be communicated in real time except on very costly networks. Fortunately, DVEs seldom require rapid change in this kind of data.

Low bandwidth operation is essential for a DVE transport layer to be immediately applicable to the currently existing infrastructure.

1.1 Other Approaches

The DVE transport layers fielded in commercial applications to date fall into two basic categories: proprietary, central server (CS) systems and those adhering to the Distributed Interactive Simulation standard (DIS) [6, 13]. CS systems are widely used as the basis for MUDs, chat rooms, and multi-user games. DIS has been widely successful as the basis for military training systems. Figure 1 rates these two approaches from the perspective of the six goals presented above on a scale of A-E. Blanks indicate goals that a given approach was never intended to address.

Both CS and DIS provide close to the best possible real-time performance and state equality. Because it uses direct peer-to-peer communication, DIS has an edge in speed. However, the use of a central server allows CS systems to do a slightly better job of providing state equality.

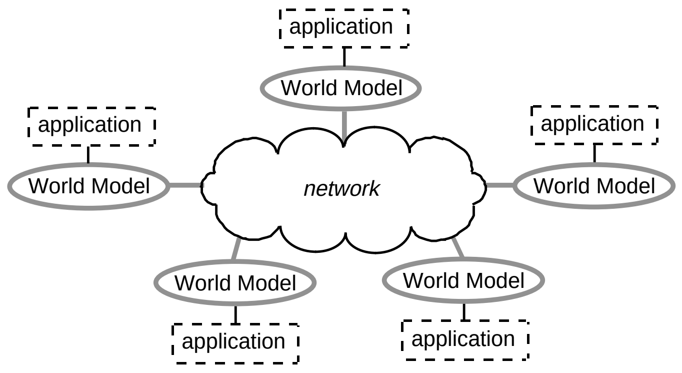

Figure 2: Object sharing in ISTP.

Even though DVEs of considerable size can be supported using a single central server, such a server acts as a bottleneck that fundamentally prohibits scalability. In contrast, DIS is fully distributed in nature and has been demonstrated to support thousands of communicating processes. However, every node in a DIS simulation is informed of every change anywhere in the simulated world. Even if the only thing a node does with most of this information is discard it, the torrent of information fundamentally limits scalability.

In contrast to DIS applications, which are essentially static, MUDs Object Oriented (MOOs) have pioneered the concept of malleable DVEs. However, since most MOOs are text-based, this work has focussed on extending behavior rather than visual models or sound effects.

Neither CS systems nor DIS make any attempt to support WWW-like operation. Rather, both approaches are oriented toward the centralized creation of applications that evolve slowly (if at all) and do not interact with each other.

A key advantage of CS systems is acceptable performance using low-bandwidth connections. In contrast, DIS-based applications typically make use of special-purpose dedicated networks.

2 Overview of ISTP

As illustrated by the last column of Figure 1, the Interactive Sharing Transfer Protocol (ISTP) is a significant improvement over CS and DIS systems in the areas where they are weak. As summarized below, this is achieved in ISTP through the combination of a number of specialized techniques.

Figure 2 shows five ISTP processes. Each process has a local copy of the data they share, which we call the *world model*. The data sharing is supported via the shared data-object model of distributed shared memory [1]. However, to provide scalability, several modifications are made in this standard approach.

To avoid writers/writers conflicts, each object in the world model has one process as its owner and only the owning process can modify it. However, the ownership of an object can be transferred from one process to another. There are no inter-process locks. Each process can separately control the times at which object-update information is sent and received.

The world model is broken into chunks called *locales* [2, 4] with disjoint sets of objects and explicit lists of neighboring locales. No process maintains a complete copy of the world model; rather, each process maintains a partial copy of only the locales it is interested in. Each process attends to several locales and interacts with the other processes that attend to any of these locales.

Like DIS, ISTP achieves near real-time interaction by using peer-to-peer UDP messaging for most updates. ISTP achieves greater scalability than DIS by associating each locale with a separate multicast communication group. (This basic idea has been used in a number of systems such as [8].)

By deciding which locales (and thus multicast groups) to attend to, ISTP processes decide which object messages to receive without experiencing any overhead for the messages they wish to ignore. The use of locales allows ISTP to support very large numbers of users and very large amounts of content, but it must be realized that the scalability is not complete. While there is virtually no limit to the number of locales that can be supported, there is definitely a limit on the total number of users and objects in any one locale at any one time. This implies that while one can model a virtual city full of people, something artificial has to be done if too many of these people gather in the same place at the same time.

For example, in a virtual football stadium, one might have to model each section of the stadium using a separate locale and restrict the spectators to interacting only with the spectators in their immediate section and its neighbors.

ISTP achieves full malleability via a hybrid communication approach. Application processes are rapidly informed of the large pieces of data they need to acquire via small objects called *links* whose principal component is a Uniform Resource Locator (URL) for the large data. The large data itself is fetched as needed using HTTP. Because this fetching is inevitably somewhat slow, careful preloading may be necessary to achieve real-time scene changes in a given DVE; however, ISTP places no limits on what can be dynamically downloaded to a process.

ISTP achieves WWW-like operation via locales, links, and a third special class of objects called *beacons* [3]. Beacons have URL-like tags and can be used as markers on other world model objects. Special ISTP servers allow the rapid retrieval of information about beacons using queries based on beacon URLs.

Beacons allow content to be easily located. Links allow it to be retrieved. Locales allow content to be encapsulated so that it can be integrated with preexisting content without accidental conflict. Someone who wishes to field new content merely has to create appropriate beacon, link, and locale objects. They do not have to interact with any central administration.

A secondary benefit of beacons is that information about them is communicated through CS-style servers, rather than via UDP multicast. Applications can use beacons to exercise tight control over state equality when this is necessary. A typical process is connected to several beacons and interacts in a controlled way with the processes owning them.

ISTP achieves fundamentally lower bandwidth operation than DIS by using reliable communication of object update messages. This allows the use of compact differential messages and eliminates the need for keep-alive messages.

2.1 User Servers

Full ISTP clients require moderately high bandwidth network connections with multicast capability. To accommodate clients with weaker network connections (e.g., modems), ISTP utilizes *user servers* as illustrated in Figure 3.

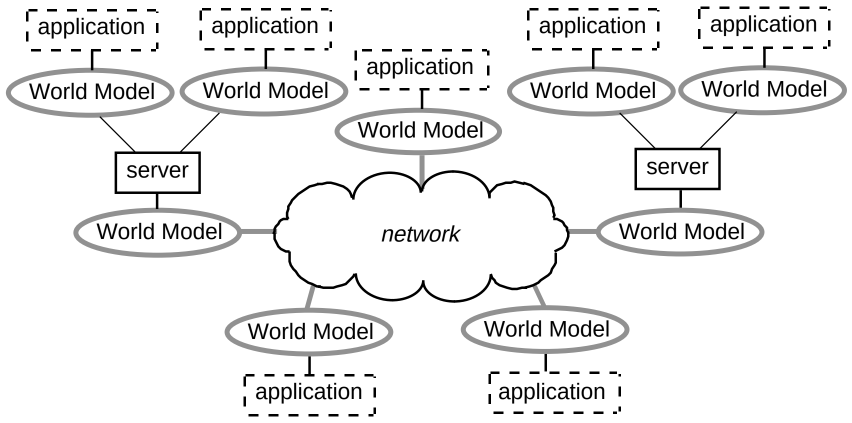

Figure 3: User servers.

A user server acts as a proxy between one or more clients and the rest of ISTP. From the perspective of the rest of ISTP, a user server looks like an application client with high-speed multicast capability, which just happens to be supporting other processes.

The primary purpose of a user server is to minimize the amount of information that needs to be communicated to a client and to do so using unicast rather than multicast. For example, in Figure 2, a given client may receive several live audio streams which it then combines into an appropriate sound image. In contrast, a user server intercepts all the sounds destined for a given client, combines them itself and then sends just one combined sound stream to the client, thereby minimizing bandwidth requirements.

The basic situation in Figure 3 is a hybrid one where clients communicate with servers via unicast, but where scalability is achieved by allowing there to be many servers who communicate with each other using multicast. This kind of hybrid approach has been suggested by a number of researchers, e.g., [7].

2.2 Architecture of ISTP

The architecture of ISTP is illustrated in Figure 4. The basic function of ISTP is to communicate objects in the shared world model underlying a DVE. It is expected that there will be many kinds of objects in this world model. However, the operation of ISTP depends only on the nine fundamental classes shown at the left of Figure 4. The dashed lines between the class names indicate multiple inheritance relationships between superclasses on the left and their subclasses on the right.

The class sp is the root of the shared world model object class hierarchy. Every object in the shared world model is an instance of a descendant of sp. The classes spLinking, spBeaconing, and spLocale correspond to the link, beacon, and locale objects discussed above. spBeacon-Monitor objects represent requests for information about beacons. An spMultilinking object is a special kind of link that acts as an index to several large pieces of data. spClass objects describe user-defined classes. spObserving objects specify which locales a process wants to receive information about. spAudioSource objects specify sources of real-time sound streams.

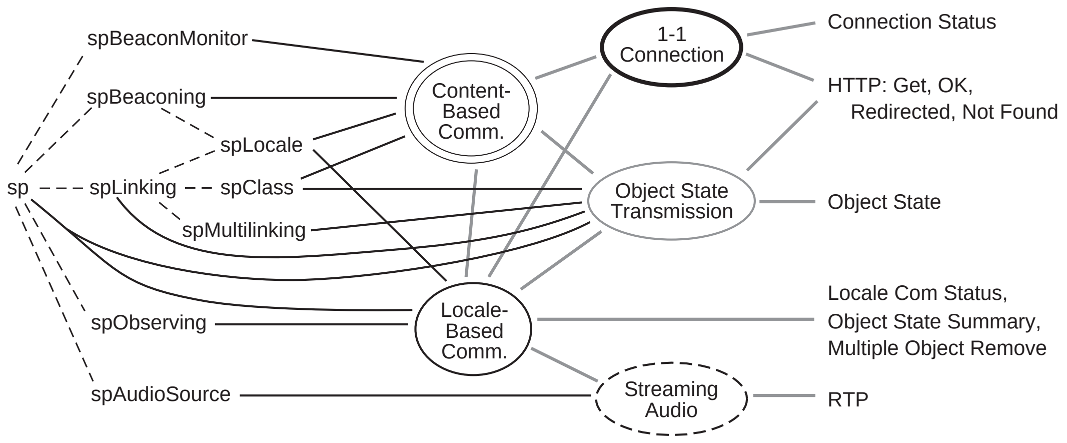

Figure 4: Object classes, subprotocols, and messages in ISTP.

The nine object classes form the interface between ISTP and the process P running on top of ISTP. All communication between P and ISTP is by means of creating, modifying, and deleting objects of these classes. Specifically, all input from P into the ISTP layer is achieved by P creating, modifying, and removing objects it owns in its local world model copy. The ISTP layer observes these changes and performs appropriate communication actions. Similarly, all output from the ISTP layer to P is achieved by the ISTP layer creating, modifying, and removing objects owned by other processes in the local world model copy. This is done based on information ISTP receives from the owning processes.

The ovals in the center of Figure 4 correspond to the five subprotocols that comprise ISTP. These subprotocols are summarized in the following sections and discussed in detail in Part III. The 1-1 Connection subprotocol is used to establish and maintain a TCP connection between two ISTP processes. The Object State Transmission subprotocol is used to communicate the state of objects from one ISTP process to another. The state messages can be sent either over a 1-1 Connection between processes or via UDP multicast.

The Streaming Audio subprotocol communicates audio data (e.g., captured by a microphone) between ISTP processes. Since it is based directly on the Real Time Protocol (RTP), little is said about the Streaming Audio subprotocol in this report. (ISTP does not currently support video streams. This could be added by creating video source objects and relying on an appropriate video stream communication protocol.)

Two higher level subprotocols are build upon the three protocols above. Locale-Based Communication is the heart of ISTP. It supports the sharing of information about objects in the world model. Content-Based Communication supports central-server-style communication of beacon information.

The solid lines in Figure 4 show which object classes are relevant to the various subprotocols. For example, Locale-Based Communication operates in the same way for every kind of object, with special meaning attached to locales and spObserving objects. In contrast, Content-Based Communication applies only to beacons, with special operations applied to locales and spClass objects.

The right end of Figure 4 lists the message types used by ISTP. Detailed specifications of the

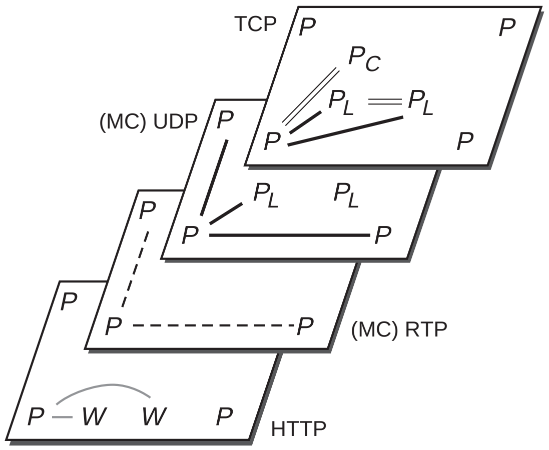

Figure 5: ISTP process interaction.

various message formats are presented in Part II. The way the messages are used is discussed in conjunction with the subprotocols that use them. The grey lines in the figure show which messages and subprotocols are used by each of the protocols. In each case, the using protocol is on the left.

The message types are the wire protocol by which ISTP process communicate with each other. All the information sent out and received by an ISTP process are in terms of these message types. Because every process is capable of being both a client and a server, every process can potentially send and receive every kind of message.

2.3 ISTP Process Interaction

The letters in Figure 5 represent the two kinds of processes that participate in an ISTP session: processes communicating using ISTP (P) and standard HTTP web servers (W).

In addition to supporting an application, any ISTP process can act as a server for Content-Based and/or Locale-Based Communication. The processes in Figure 5 that are providing these services are indicated using subscripts—P_C and P_L respectively.

A key feature of ISTP is that it is completely distributed with no centralized control process. All ISTP processes are capable of participating in the entire protocol and thus can be ISTP servers. However, at any given moment, it is likely that only a minority of ISTP processes are in fact acting as servers. Furthermore, there might well be special ISTP implementations that are tuned to be good servers. Therefore, from the perspective of an application, it is natural to think of a distinction between clients and servers even though ISTP makes no such distinction.

ISTP is a hybrid protocol that builds on top of four underlying protocols: TCP, UDP, RTP, and HTTP. This relationship is illustrated by the four horizontal planes in Figure 5. The appearance of a letter in the same position in successive planes indicates that the process participates in communication using multiple underlying protocols.

TCP is used for the reliable communication of control information. UDP and RTP are used for the communication of time critical information. UDP and RTP messages are both sent via multicast whenever possible. HTTP is used for the distribution of large pieces of data.

The lines in Figure 5 represent the various kinds of interactions that the ISTP process in the lower left participates in. The ISTP subprotocol being used by a path is indicated by drawing its line in Figure 5 using the same style (solid, double, dashed or grey) as the oval around the subprotocol in Figure 4.

Just as all the web servers in existence are implicitly combined into a single world wide web, all the ISTP processes that are running at any given moment are capable of interacting with each other. Running a web browser and knowing a Uniform Resource Locator (URL) (or running a web server and advertising a URL) are the only things that are needed to participate in the world wide web. Similarly, running an ISTP process and knowing (or advertising) a beacon tag are the only things that are necessary to enter the ISTP world.

3 Object State Transmission

World model objects have instance variables, some of which are shared, while others are used for storing local auxiliary data. Objects are communicated in ISTP by creating Object State messages. These messages are sometimes sent over 1-1 Connections using TCP and sometimes sent using multicast UDP.

An Object State message contains the description of the current state of the shared variables of one or more objects. When an Object State message is received by another process, it is used to create (or update or remove) a copy of the objects described.

Three different kinds of descriptions are used to describe objects: full descriptions, which describe every shared value; differential descriptions, which describe only changed values in shared objects; and link differential descriptions, which specify how the data associated with a link has changed. Differential descriptions are much more compact than full descriptions, but their interpretation depends on the previous receipt of a full description. Therefore, there are some situations where full descriptions must be sent.

The communication of object state proceeds in the same way for every kind of object. However, two classes of objects are worthy of special note: *links* and *class descriptors*.

When using UDP multicast, the size of an Object State message is limited so that it can fit in a single UDP packet. This places a relatively small upper limit on the size of the description for any one object. To accommodate inherently large objects, link objects break the associated data into two parts.

A link object contains a URL and acts as a handle for a large piece of data. The associated large data is obtained using standard HTTP Gets and replies, taking advantage of caching whenever possible. Differential link descriptions make it possible to make small rapid changes to the data corresponding to a link without having to refetch the data in full.

Among other things, An spClass class object specifies the layout of the instance variables for an object class. ISTP is able to transmit arbitrary objects in messages with reference to these specifications.

For most types of instance variables (e.g., numbers, arrays of numbers, and strings) composing and decomposing object descriptions is straightforward. However, two kinds of values are worthy of particular note: *references to other objects* and *times*.

In descriptions, ISTP uses Globally Unique IDs (GUIDs) to refer to objects. (It is expected that for speed in a given process one might use pointers to refer to objects, but this is outside the scope of ISTP.)

To allow GUIDs that are unique in time and space to be generated without reference to any central server, Internet addresses are used as the basis for ISTP's GUIDs. This requires the use of a large number of bits per GUID—a 32-bit Internet address identifying a machine plus a 64-bit intra-machine unique value.

Given the large size of the GUIDs, it is important that ISTP can compress the GUIDs in typical Object State messages so that, on average, they require many fewer than 96 bits each. This is possible because the typical message contains many GUIDs that were all generated by the same process at approximately the same time and therefore most of the bits in these GUIDs are the same.

To allow precise specification of event timing, ISTP uses compact absolute time stamps. As discussed in [9], ISTP incorporates methods for adjusting the time values that are communicated between machines so that they are meaningful on each machine even when the clocks on the various machines are not synchronized.

Whenever an object description is transmitted, the associated class (and superclasses) descriptors must be communicated as well, unless it is known that the recipient already knows these classes. This is necessary so that the recipient can decode the description of the object.

Using ISTP, it is easy to arrange things so that communicating spClass objects consumes a negligible percentage of bandwidth during Locale-Based Communication. However, a moderate percentage of bandwidth is required for this during Content-Based Communication. Fortunately, Content-Based Communication as a whole typically consumes only a very small part of the total bandwidth used by ISTP.

4 Locale-Based Communication

The Locale-Based Communication subprotocol of ISTP transmits information about changes in the world model using UDP multicast backed up by TCP unicast when UDP messages are lost. This approach is optimized for minimal latency communication to an audience of interested processes, where interest is defined in terms of the 'locality' of the objects.

In ISTP, 'locality' is specified explicitly using locale objects [4] rather than geometrically. Each object is explicitly designated to be in one locale and each locale explicitly specifies which other locales are its immediate neighbors. (Using explicit rather than implicit specifications of locality and neighborhood allows greater flexibility.) If a process owns an object in locale L, then it sends out messages about changes in the object via L.

A process indicates which locales it is interested in obtaining information about by creating spObserving objects that specify the point of view of the process. In general, an spObserving object in a locale L indicates interest in L and the immediate neighbors of L. A process receives information about all the objects in each locale that an spObserving object it owns indicates interest in.

Each locale has a Locale-Based Communication server associated with it, with a given server possibly supporting many locales. The mapping of locales to servers is specified using Internet addresses in locale objects. This allows an application to exercise complete (dynamic) control over where the locales it creates are served. Scalability to arbitrarily many locales is achieved by duplicating Locale-Based Communication servers so that no one server has more work than it can handle.

The server for a locale is responsible for selecting the communication addresses to use for the locale. One address is used for messages describing changes in the objects in the locale. Another address is used for streaming audio communication. It is intended that the communication addresses be multicast addresses. However, ISTP supports simulated multicasting when true multicasting is not available.

4.1 Joining a Communication Group

If it wants to join the communication group for a locale L, an ISTP process P opens a 1-1 Connection to the Locale-Based Communication server S for L and then sends a Locale Com Status message to S requesting inclusion in the communication group. Among other things, the Locale Com Status message specifies whether P wants an initial download of the current state of all objects in the locale.

If S decides to allow P to join, then it replies with a Locale Com Status message specifying the proper addresses to use for reading and writing. (S monitors all the message traffic on this address in order to maintain an up-to-date snapshot of the current state of every object in L.)

If requested by P, S also sends an Object State message specifying the state of every object in L so that P can initialize its world model. This message also contains information about the locales that neighbor L.

The neighbor information is communicated in case P knows about L without yet knowing about the neighbors of L. It is possible that S may not itself serve a given neighbor L' of L. In that case, S uses the Content-Based Communication protocol to obtain information about L' from the server S' that serves L'. (This is possible because locales are a special kind of beacon.)

When P receives the proper communication address to use for writing or reading, P opens a direct connection to the address. This allows the process to send/receive information about objects in the locale with the least possible latency.

If P is unable to achieve a multicast connection to the other processes in a group (i.e., because part of the intervening network does not support multicast) then ISTP simulates a multicast connection to P via the 1-1 Connection to S. This increases the latency to and from P because messages have to pass through S rather than going directly to and from other processes, but is otherwise transparent to the processes in the communication group.

If interference develops from non-ISTP traffic on a multicast communication address, the server changes to a different address, informing the various participants of the change using Locale Com Status messages.

To terminate interaction via L, P can simply stop communicating and terminate its connection to S after making sure that S has received up-to-date information about any objects owned by P that were in L.

An additional responsibility of S is to monitor (via Connection Status messages) the 1-1 Connection to P to determine whether P crashes. If the connection to P is broken, then S can take some appropriate action such as sending a Multiple Object Remove message to the other processes in the group informing them that they should remove the objects owned by P from their world model copies. This removes all trace of the defunct process.

4.2 Object Updates

ISTP's object state communication approach is a compromise between speed and reliability. For speed, Object State messages are sent using peer-to-peer UDP multicast, with one Object State message (containing as many object descriptions as possible) in each UDP packet. (Because there is a one to one correspondence between UDP packets and Object State messages, the terms 'message' and 'packet' are used interchangeably below.)

To ensure that the Object State messages sent by a process P via a locale L can be decoded by the receivers, P must ensure that the receivers know the classes of all the objects described. This can be done by including the relevant classes in the messages sent. However, it can be done much more efficiently, by ensuring that if L contains an object of class C, then L also includes the class descriptor for C (and any necessary superclasses) if they are not built in. This is sufficient, because Locale-Based Communication ensures that each receiver always knows about every object in a given locale, rather than just some of them.

The use of UDP minimizes latency when messages succeed in arriving, but it is possible for messages to arrive out of order, more than once, or not at all. For reliability, TCP communication with the appropriate Locale-Based Communication server is used as a backup to obtain information contained in lost and excessively delayed UDP messages.

To maximize interactivity, UDP messages are processed as soon as they arrive. To allow for proper handling of out-of-order messages, each object state description contains a 16-bit counter value that is incremented by one each time the state of the object changes. These values can be used to reject duplicate descriptions and enforce ordering constraints (where necessary) on a per-object basis. It would be a waste for ISTP to force total ordering of UDP Object State messages themselves, since there is no problem in handling updates to different objects out of order.

The only constraint that ISTP applies at the UDP level is enforcing a time limit MaxDelay (on the order of a few seconds) on how late a message can arrive. Specifically, any Object State message whose lateness of arrival (determined using a timestamp in the message) is greater than MaxDelay is discarded.

Without some forced time limit, there would be no bound on how late an out-of-order message could arrive. This would cause several kinds of problems. For instance, when an object is removed, a late arriving, out-of-order Object State message about the object could cause the object to erroneously reappear.

To guarantee that an ISTP process can determine when one of the UDP Object State messages sent to it has been lost, the Locale-Based Communication server S for a locale L sends out periodic Object State Summary messages specifying the current counter values for objects that have been updated since the last Object State Summary message was sent. This alerts processes to information that has been lost in transmission. Object State Summary messages are transmitted reliably (on the order of once every few seconds) over the 1-1 Connections between the server and the processes doing I/O in L.

If P notices that it does not have up-to-date information about some object, P asks S for the latest information. Both the query and S's reply are sent via TCP. As a result, it can be guaranteed that P will eventually have up-to-date information.

If a process P notices that information it sent by UDP failed to reach S, then P resends the current state of the relevant objects via TCP over P's 1-1 Connection to S.

A basic principle at work above is that reliability should, as much as possible, be introduced at a high level where one can take maximum advantage of the constraints of the particular domain; rather than by using brute force at a low level. In particular, an Object State message that is lost but soon rendered obsolete by subsequent Object State messages does not have to be resent. The approach used by ISTP guarantees reliable delivery while ensuring that effort is not wasted in such situations.

5 Content-Based Communication

As a counterpoint to the location-specific nature of Locale-Based Communication, Content-Based Communication supports content addressable connections between ISTP processes. This approach is optimized for reliable communication of information to an audience of interested processes, where interest is defined in terms of explicit requests for beacons.

Applications create beacon objects with URL-like tags. ISTP processes providing Content-Based Communication service are used to store information about every beacon object in existence. Scalability is achieved by partitioning the space of beacons between many Content-Based Communication servers. The mapping of beacons to servers is done based on the DNS-name portion of beacon tags. This allows an application to exercise complete control over where the beacons it creates are served.

Beacon queries are represented by objects called spBeaconMonitors. An spBeaconMonitor specifies a pattern, which is a URL that can contain wild card characters. The pattern specifies which beacons match the query. The DNS name part of an spBeaconMonitor pattern must be fully specified. It is used to select the Content-Based Communication server that can supply the answer.

Whenever a process P makes (or modifies) a beacon (or spBeaconMonitor) B, it opens a 1-1 Connection to the Content-Based Communication server S specified by the tag of B (if a connection is not already open). P then uses the Object State Transmission protocol to communicate the current state of B to S. So that S can properly understand the description of B, P includes a description of the spClass object for B (if the class is not built-in). P also includes a description of the spLocale object B is in.

(Since classes and locales are link objects, their descriptions in Object State messages are small. Also, P is permitted to omit the class and/or locale, if it can be sure that S has already been informed of these objects and is retaining them. For example, when B has been modified and P sent the class and locale in a previous message.)

When S receives the description of a beacon B it serves, it updates its world model copy. In response to an spBeaconMonitor, S replies to P with descriptions of all the beacons known to S that match the pattern. This reply is done in one Object State message so that P can know when the reply is complete. The reply includes the classes of the beacons in question and the locales the beacons are in. The locales are included so that P can connect to them if desired, in order to obtain information about other nearby objects.

Like the world wide web, the ISTP world is a single implicit entity spanning the globe. To enter this world, all you have to do is run an ISTP process. However, merely running such a process does not by itself allow you to observe any world model objects other than the ones you create yourself. To get connected to other processes, an ISTP process must first locate one or more locales (via beacons) and then connect to these locales. Once this has been accomplished, the process can participate in the portion of the ISTP world that is contained in these locales.

6 1-1 Connection

The 1-1 Connection subprotocol of ISTP creates a TCP connection between two processes, which is used for the communication of control information. In general, these connections are between an ordinary ISTP process P and a Content- or Locale-Based Communication server process S.

To create a 1-1 Connection to S, P first opens a TCP connection to S and then sends an HTTP Get message that requests an ISTP connection. S then replies with an ISTP Connection Status message with the status 'Initialize', to indicate that the connection is open.

After the initial connection, and in the absence of other message traffic, P and S each send periodic Connection Status messages specifying that the connection remains open. Either party can send a 'Close' Connection Status message specifying that the connection should be terminated. In addition, either party can send another 'Initialize' Connection Status message at any time specifying that the other party should act as if the connection had just been opened and resend all pertinent information. This is convenient when one of the processes associated with a connection has to be reinitialized or restarted.

It is not possible to connect with a given part of the ISTP world without first identifying the relevant locale (via a Content-Based Communication server) and then initiating communication in the locale (via a Locale-Based Communication server). As a result, content-based and/or Locale-Based Communication servers can exercise complete control over what processes can participate in which parts of the ISTP world by choosing which 1-1 Connections to accept and reject.

In particular, instead of accepting a 1-1 Connection as outlined above, S can reject it. This is done by sending an HTTP reply with status 'Not Found' instead of an ISTP Connection Status message. Alternately, S could choose to accept the connection, but screen the information subsequently sent to P.

A final option available to S is to send an HTTP reply with status 'Redirected' specifying the URL of a different process to connect to. This allows a process to act as a dispatcher forwarding requests to a group of other processes that provide services.

For example, a very large application might wish to maintain replicated Locale-Based Communication servers either to do dynamic load balancing or to reduce latency by connecting users to servers geographically near them. This can be done by having a centralized dispatching process that selects which server to connect a given user to. Such a dispatching process could also monitor the replicated Locale-Based Communication servers to assess the loads on them and restart any servers that crash.

HTTP messages are used for initiation, rejection and forwarding in the 1-1 Connection subprotocol of ISTP so that ordinary HTTP web servers can be used for access control and dispatching.

All services are provided via connections initiated by the processes requesting the services so that communication can proceed even if the requesting process is inside a firewall. In general, ISTP processes providing services must run outside firewalls.

Although not yet designed in detail, security could be supported in ISTP via encryption of the messages sent. The administration of multicast encryption keys could be handled by Locale-Based Communication servers.

7 Spline

Like TCP or DIS, ISTP is just a communication protocol. Much more than than ISTP is needed to make it easy to write DVEs. Using ISTP as a basis, a prototype implementation of a full DVE platform called the Scalable Platform for Large Interactive Network Environments (Spline) [11] has been implemented. Several applications are being constructed on top of Spline [11]. However, much more experimentation has to be done before one could consider ISTP a success. We hope that suitably tempered by experience, ISTP can grow into a candidate DVE communication standard [5].

Acknowledgments

The design and implementation of Spline and ISTP is the result of a four-year effort by a large group of people. Richard Waters and David Anderson have been the principle architects of Spline and ISTP from their earliest inception. In addition, they were the principal implementors of the initial version of Spline (Spline 1.5) and led the reimplemention effort that created Spline 3.0.

Major contributions to the design and implementation of Spline and ISTP were made by John Barrus. In particular, he participated in the design from the very beginning and was a co-designer of locales.

A large team of people participated in the implementation of Spline 3.0 under the leadership of Waters and Anderson, who were principally responsible for the design. William Lambert took over leadership of the overall Spline effort at the end of 1996. William Yerazunis managed the software engineering aspects of the project and implemented smooth motion operations. Derek Schwenke implemented the system core including much of ISTP and the visual renderer. He took over management of the software engineering effort in mid 1997. Sam Shipman implemented the Java interface and the audio renderer. Alex Greysukh implemented the underlying network communication code, building on the work of Hiroshi Kozuka and Vu Phan. Rob Kooper and David Ratajczak implemented the routines for manipulating transformations. Christina Fyock, Evan Suits, and Barry Perlman created the first Spline 3.0 applications.

References

[1] H.E. Bal, *Programming Distributed Systems*, Prentice Hall International, 1991.

[2] Barrus J.W., Waters R.C., & Anderson D.B. (1996) "Locales and Beacons: Efficient and Precise Support for Large Multi-User Virtual Environments", *IEEE Virtual Reality Annual International Symposium*, (Santa Clara CA, March 1996), 204–213, IEEE Computer Society Press, Los Alamitos CA.

[3] J.W. Barrus, R.C. Waters, & D.B. Anderson, *Locales and Beacons: Efficient and Precise Support for Large Multi-User Virtual Environments*, technical report 95-16a, MERL Cambridge MA, August 1996.

[4] J.W. Barrus, R.C. Waters, & D.B. Anderson, "Locales: Supporting large multiuser virtual environments", *IEEE Computer Graphics and Applications*, 16(6):50–57, November 1996.

[5] D. Brutzman, M. Zyda, K. Watsen, & M. Macedonia, "Virtual Reality Transfer Protocol (VRTP) Design Rationale", web document "http://www.stl.nps.navy.mil/˜brutzman/vrml/vrtp_design.ps", January 1997.

[6] Calvin J., et al (1993) "The SIMNET Virtual World Architecture", *Proc. IEEE Virtual Reality Annual International Symposium*, 450–455, (Seattle WA, Sept. 1993).

[7] R. Lea, Y. Honda, K. Matsuda, & S. Matsuda, "Community Place: Architecture and Performance", *VRML '97 Symposium*, (Monterey CA, February 1997).

[8] M.R. Macedonia, et al, "Exploiting Reality with Multicast Groups", *IEEE Computer Graphics and Applications*, 15(5):38–45, September 1995.

[9] R.C. Waters, *Time Synchronization In Spline*, technical report 96-09, MERL Cambridge MA, April 1996.

[10] Waters R.C. & Barrus J.W., "The Rise of Shared Virtual Environments," *IEEE Spectrum*, 34(3):20-25, March 1997.

[11] R.C. Waters, D.B. Anderson, J.W. Barrus, D.C. Brogan, M.A. Casey, S.G. McKeown, T. Nitta, I.B. Sterns, & W.S. Yerazunis, "Diamond Park and Spline: A Social Virtual Reality System with 3D Animation, Spoken Interaction, and Runtime Modifiability," *Presence: Teleoperators and Virtual Environments*, 6(4):461–480, August 1997.

[12] Waters R.C., Anderson D.B., Greysukh A., Lambert W., Kozuka H., Perlman B., Phan V., Schwenke D., Shipman S., Suits E., and Yerazunis W., *The ANSI C (Internal) Spline Version 3.0 Application Program Interface*, MERL TR 97-11, October 1996.

[13] *Standard for Information Technology, Protocols for Distributed Interactive Simulation* (DIS ANSI/IEEE standard 1278-1993), American National Standards Institute, 1993.

Part II
ISTP Message Types

This second part of the report specifies the individual messages types in ISTP. It is a complete specification of what each bit in each message means. The context of use of the messages is explained in Part III.

The formats described are the formats of messages on the network. All data is in network byte order using IEEE floating point numbers. An implementation of ISTP (e.g., on PCs) might well use an in-memory representation with a different byte order and/or layout of fields.

8 HTTP Get

The ISTP protocol incorporates several simple HyperText Transfer Protocol (HTTP) messages. In particular, it includes HTTP Get messages and several kinds of HTTP replies. These messages are used so that standard HTTP web servers can participate in ISTP sessions as sources of link data and dispatchers to ISTP servers.

HTTP messages are ASCII strings with fields delimited by line breaks. These can take a variety of forms and contain a variety of optional fields. However, only a few selected forms are used by ISTP.

The format of the HTTP Get messages used by ISTP is shown in Figure 6. The `"GET"` must be capitalized and the message is terminated by two consecutive line breaks. The meaning of the *locator* is discussed below.

ISTP requires that `"HTTP/1.0"` appear on the first line even though this is optional in HTTP. The reason for this is that this causes the replies to be more easily understood by ISTP. (All the HTTP Get messages used by ISTP originate from ISTP processes, rather than web servers.)

There are additional fields that can appear in an HTTP Get message such as User-Agent. These fields are permitted in ISTP-generated GET messages when used in standard ways.

Getting link data. HTTP Get messages are used in two different ways in ISTP. First, they are used in an entirely standard way as part of the Object State Transmission subprotocol of ISTP to obtain the data corresponding to link objects, see Section 20.2.

For example, suppose `"HTTP://`*DNSname*`:`*port*`/`*locator*`"` is the Uniform Resource Locator (URL) in a link object X. (The port number can be omitted in which case it defaults to 80.)

To fetch the data corresponding to X, an ISTP process P first opens a two-way TCP connection to the process W running on the machine identified by *DNSname* that responds on port *port*. Opening this involves the exchange of IP messages. These messages are implicitly part of ISTP; however, they are entirely standard in form and are therefore not discussed here.

Once the connection is open, P sends an HTTP Get message of the following form. Note

```
GET /locator HTTP/1.0
```

Figure 6: Format of an HTTP Get message.

that the *DNSname* and *port* used to refer to *W* are not part of the HTTP Get message and therefore cannot be used by *W* when deciding on its response.

```
GET /locator HTTP/1.0
```

More specifically, if the URL were `"HTTP://www.merl.com/models/big-room.vrml"`, then *P* would connect to the web server *W* for `www.merl.com` and send the following request.

```
GET /models/big-room.vrml HTTP/1.0
```

A web server (or ISTP processes) *W* that receives an HTTP Get like the one above will respond with one of the HTTP responses discussed in the next three sections. These either provide the requested data, suggest an alternate place to find the data, or state that the data is not available.

Requesting 1-1 connections. The second way that ISTP uses HTTP Get messages is when requesting that a 1-1 connection be made to a Content- or Locale-Based Communication server, see Section 19.

When an ISTP process *P* wishes to open a 1-1 connection, the machine and port to connect to are specified by a beacon or locale object. As above, *P* starts by opening a two-way TCP connection to the appropriate process *S*. *P* then sends an HTTP Get where the *locator* specifies the kind of server it wishes to connect to. In particular, the *locator* must be either `"ISTP-content-based-server"` or `"ISTP-locale-based-server"`. For example, *P* might send:

```
GET /ISTP-content-based-server HTTP/1.0
```

If *S* agrees to open the requested 1-1 connection, it replies with a Connection Status message, see Section 13, with a Status of Initialize. Otherwise, *S* might send either of the three replies discussed in the following sections.

If *S* replies with an HTTP OK Reply, this indicates that *S* is actually a web server rather than an ISTP server and that it happens to have content that matches the *locator* specified. This is an erroneous situation, which indicates the failure to establish an ISTP 1-1 connection. *P* abandons any further attempt to create the desired connection. (The *locators* used to open 1-1 connection are chosen to be ones that are unlikely to accidentally trigger the receipt of information from an ordinary web server.)

If *S* replies with an HTTP Redirected Reply, then *P* tries again to establish the 1-1 connection using the DNS-name and port specified in the reply. This is used when *S* wants to act as a dispatcher controlling which actual server process *P* gets connected to. HTTP messages are used for this so that an ordinary web server can act as such a dispatcher.

If *S* replies with an HTTP Not Found Reply, then this indicates that *S* refuses to allow the connection. In that case, *P* could try again after some period of time. Alternatively, the reply may indicate an erroneous situation where *S* is actually an ordinary web server, that does not have content that matches the *locator*.

9 HTTP OK Reply

HTTP OK Reply messages are used by web servers to reply with requested data. An OK Reply in response to an HTTP Get with the format shown in Figure 6 has the form shown in Figure 7. (Replies generated by other kinds of HTTP Get requests are not relevant to ISTP.)

```
HTTP/1.0 2xx ...
...
```

Figure 7: Format of an HTTP OK Reply message.

The number 2xx is a three digit condition code. The fact that the first digit is 2 indicates that the HTTP Get was successful and the reply contains the requested data. There are a number of optional fields that may be present in the body of the reply. None of these are required by ISTP, but their presence must be tolerated. The data contained in the reply begins immediately after the first pair of consecutive line breaks. The following is an example of an OK reply.

```
HTTP/1.0 200 Document follows
Server: CERN/3.0A
Date: Mon, 21 Jul 1997 14:12:51 GMT
Content-Type: text/html
Content-Length: 5416

<html>
<head>
<title>MERL--A Mitsubishi Electric Research Laboratory</title>
...
```

ISTP processes never send HTTP OK Reply messages, but they receive them from ordinary web servers. This happens as part of the Object State Transmission subprotocol of ISTP when obtaining the data corresponding to a link object, see Section 20.2.

As noted in the previous section, it is also possible for an ISTP processes to receive an HTTP OK Reply as part of an erroneous attempt to open a 1-1 connection to an ISTP server.

10 HTTP Redirected Reply

HTTP Redirected Reply messages are used to indicate an alternate location at which to satisfy an HTTP Get request. When given in response to an HTTP Get with the format shown in Figure 6, an HTTP Redirected Reply has the form shown in Figure 8. (Replies generated by other kinds of HTTP Get requests are not relevant to ISTP.)

The number 3xx is a three digit condition code. The fact that the first digit is 3 indicates that redirection is being specified. A Redirected Reply contains a field called 'Location', which is followed by an alternate URL to use. There are a number of other optional fields that may

```
HTTP/1.0 3xx ...
...
Location: alternate-URL
...
```

Figure 8: Format of an HTTP Redirected Reply message.

```
HTTP/1.0 404 ...
...
```

Figure 9: Format of an HTTP Not Found Reply message.

be present in the body of the reply. None of these are required by ISTP, but their presence must be tolerated. The following is an example of a Redirected reply.

```
HTTP/1.0 302 Object moved
Server: CERN/3.0A
Date: Mon, 21 Jul 1997 16:12:51 GMT
Location: http://www.merl.com/obsolete/models/big-room.vrml
Content-Length: 198
...
```

Getting link data. HTTP Redirected Reply messages are used in two different ways in ISTP. First, they are used in an entirely standard way as part of the Object State Transmission subprotocol of ISTP when obtaining the data corresponding to link objects, see Section 20.2.

For example, the reply above might be received in response to the request below. Upon getting such a reply, the Object State Transmission subprotocol should proceed to get the desired data from the specified alternate location.

```
GET /models/big-room.vrml HTTP/1.0
```

Requesting 1-1 connections. The second way that ISTP uses HTTP Redirected Reply messages is when requesting that a 1-1 connection be made to a Content- or Locale-Based Communication server, see Section 19. If the response to a request for connection is a Redirected Reply, then the 1-1 Connection subprotocol proceeds by trying to make the connection to the specified alternate location.

When creating 1-1 connections, HTTP Redirected Reply messages are supported so that a dispatching process can select which of a group of servers to use to satisfy a given request. Standard HTTP messages are used so that this dispatcher can be a standard HTTP web server.

11 HTTP Not Found Reply

HTTP Not Found Reply messages are used to indicate that an HTTP Get request cannot be satisfied. When given in response to an HTTP Get with the format shown in Figure 6, an HTTP Not Found Reply has the form shown in Figure 9. (Replies generated by other kinds of HTTP Get requests are not relevant to ISTP.)

The number 404 is a three digit condition code indicating that the requested data is not available. (Other codes beginning with 4 and 5 indicate other error conditions.) There are a number of other optional fields that may be present in the body of the reply. None of these are required by ISTP, but their presence must be tolerated. The following is an example of a Not Found reply.

MessageType 12 bits - indicates type of message.
Length 20 bits - total number of bytes in message.
SendTime 32 bits - time message was sent in milliseconds modulo one week.
TopicID 32 bits - compressed GUID identifying subject of message.
NumberOfProcessIDs 16 bits - number of prefixes G in ProcessIDTable.
ProcessIDTable G index/ProcessID pairs (16+80 bits each).

Figure 10: Initial fields in an ISTP message.

```
HTTP/1.0 404 Not found - file doesn't exist or is read protected ...
Server: CERN/3.0A
Date: Mon, 21 Jul 1997 16:33:26 GMT
Content-Type: text/html
Content-Length: 232
...
```

Getting link data. HTTP Not Found Reply messages are used in two different ways in ISTP. First, they are used in an entirely standard way as part of the Object State Transmission subprotocol of ISTP when obtaining the data corresponding to link objects, see Section 20.2.

For example, the reply above might be received in response to the request below. Upon getting such a reply, the Object State Transmission subprotocol gives up trying to get the data corresponding to a link object.

```
GET /models/big-room.vrml HTTP/1.0
```

Requesting 1-1 connections. The second way that ISTP uses HTTP Not Found Reply messages is when requesting that a 1-1 connection be made to a Content- or Locale-Based Communication server, see Section 19. If the response to a request for connection is a Not Found Reply, then this indicates that the target process is either unwilling or unable to create the requested connection. The requesting processes may attempt to establish the connection again at a later time, but there is no guarantee that it will succeed.

12 ISTP Message Header

Every ISTP message that is not also an HTTP message begins with the fields shown in Figure 10. The MessageType specifies which kind of ISTP message a given message is. Twelve bits are provided so that additional types can be used to support added messages and revised versions of the ISTP protocol without confusions with the initial version described here.

MessageTypes whose first 8 bits are 71 = 0x47 = 'G' or 72 = 0x48 = 'H' are prohibited so that HTTP Get and reply messages cannot be confused with other ISTP messages. All other MessageTypes are permitted. (In particular, there is no assumption that MessageTypes correspond to ASCII characters.)

The Length specifies the total number of bytes in the message. This is useful because several ISTP messages are variable in length.

The SendTime is the time in milliseconds modulo one week (604,800,000 msecs) that the message was sent. The SendTime is set by the sending process based on the clock in the sending processes. No assumptions are made about time synchronization between the clocks on different machines.

SendTimes are used to estimate time differences between machines (see Section 13.1) and to determine whether messages have been received out of order or excessively late. The way times are processed is discussed in a separate subsection below.

The TopicID is a GUID (see Section 12.2) that identifies a key object that the message is about. The object this GUID refers to is different in nature for each type of message.

The NumberOfProcessIDs specifies the number of entries in the ProcessIDTable.

The ProcessIDTable contains entries that allow for the compact representation of GUIDs in the rest of the message, including the TopicID. Each entry contains a 16-bit index indicator and an 80-bit ProcessID (see Section 12.2).

There is no explicit support for error detection/rejection/correction within ISTP messages, because ISTP relies on the underlying UDP and TCP protocols for this.

12.1 Time

As noted above, times in ISTP are represented in milliseconds modulo one week. Milliseconds is used as the units, because this allows reasonably high precision while still allowing 32-bit times to represent a reasonably long time span.

The modulus of 604,800,000 was chosen because it has a number of advantages. To start with, the modulus is easily understood in human terms. In addition, it is less than 2^31, so that negative values can be represented and two times can be added or subtracted without overflow problems. This makes computations involving the times more efficient. Further, the modulus is greater than 2^30 which means the 32 bits used to represent a time are used relatively efficiently. Lastly, the modulus has many divisors, which is convenient in some situations (see [9]).

$$604,800,000 = 2^{10} \times 3^3 \times 5^5 \times 7$$

The fact that times are represented using modular arithmetic complicates the comparison and manipulation of times, but allows them to be represented using 32 rather than 48 or 64 bits. In particular, to compare two times, it must be assumed that the do not represent actual times that differ by more than half the modulus (i.e., 3.5 days). This is a reasonable assumption given that the basic time scale of interest to ISTP is seconds.

In the following, the discussion often speaks of adding, subtracting, and comparing time values. It should be realized that in each of these situations, proper modular operations must be used. (As discussed in [9], one efficient option is to convert all times into what is effectively an absolute representation for internal processing, and use a modular representation only in messages.)

ISTP does not assume that the clocks in a set of communication machines are synchronized. This is important because the machines may belong to different organizations and therefore it may not be possible for their clocks to be synchronized. ISTP has to estimate the degree to which clocks are not synchronized in situations where this matters (see Section 13.1). Note that being able to represent times that differ by days, makes it possible to accommodate clocks that are out of synchronization by a day or more.

12.2 GUIDs

Every object transmitted by ISTP is identified by a Globally Unique Identifier (GUID). ISTP uses 96-bit (12-byte, 3-word) GUIDs that are unique in time and space. (These will expand to 192 bits under version 6 of the Internet Protocol (IPv6).) The GUIDs are composed of two parts: a ProcessID and an ObjectID.

ProcessID 80 bits - identifies ISTP process. (176 bits under IPv6.)
ObjectID 16 bits - identifies object within process.

A ProcessID identifies an ISTP process. This value is assigned whenever a new process starts and is guaranteed to be unique in space and time (for a century). This value is opaque. No way is specified for obtaining any information about a process if one only has a ProcessID. The ProcessID zero is reserved for indicating built-in objects—i.e., built-in classes.

An ObjectID consist of 16 bits and uniquely identifies an object within a single process. As an ISTP process creates new objects, it generates GUIDs for them by changing the ObjectID while holding the ProcessID constant. GUIDs are never reused. Once 2^{16} GUIDs have been generated, an additional ProcessID is created for use by the process.

The ProcessID zero combined with an ObjectID of zero represents a reference to no object. It is used when a field that may point to an object, but is not required to, does not point to any object.

Compressed GUIDs. To promote memory and communication efficiency, GUIDs are represented in ISTP messages in a compressed form consisting of a 16-bit ProcessIDTable index and the 16-bit ObjectID.

A ProcessIDTable of ProcessIDs is used to decode the ProcessIDTable indexes in compressed GUIDs. The ProcessIDTable index zero is reserved for referring to the ProcessID zero. This is implicit and does not require any space in the ProcessIDTable in a given message. A compressed GUID that is all zero is a reference to no object.

For example, an Object State Message containing descriptions of 3 objects, might contain the following 6 GUIDs—1 being the TopicId of the message, 3 being the IDs of the objects, and 2 being contained in fields in the objects. (The example is merely illustrative, showing 24 decimal digits corresponding to the 80-bit ProcessIDs and 5 decimal digits corresponding to the 16-bit ObjectIDs.)

```
876590013698865982315978 42876   876590013698865982315978 36834
876590013698865982315978 36782   876590013698865982315978 36138
876590013698865982315978 36788   100088634500832640463099 16458
```

A key thing to notice about these GUIDs is that 5 of the six were created by the same process and have the same ProcessID. As a result, these GUIDs would be represented in the Object state message using a 2-entry ProcessIDTable like the following.

```
23 876590013698865982315978
88 100088634500832640463099
```

Using explicit index values in the ProcessIDTable instead just using consecutive small integers that do not need to be explicitly stored in the table increases the size of the table, but makes it easier to construct the table. If the indexes were required to be consecutive small integers, then the index values would have to be recomputed each time a message was created.

Using the table above, the six GUIDs would be represented as follows.

```
23 42876   23 36834
23 36782   23 36138
23 36788   88 16458
```

(The way GUIDs are represented within a single process is not part of the ISTP specification. However, it is suggested that compressed GUIDs also be used, with reference to an in-memory ProcessIDTable. This table could be represented compactly, eliminating the need for explicit representation of index values in the table. Note that the size of this table is not a limit on the number of GUIDs simultaneously known to a process, but only a limit on the number of ProcessIDs simultaneously known to a process. A sparse table representation is used in ISTP messages so that ISTP messages can be rapidly constructed based on a compact in-memory ProcessIDTable.)

GUID compression ratios. The value of the ProcessIDTable in an ISTP message is that common ISTP messages (e.g., Object State messages and Object State Summary messages) contain many GUIDs. In one of these messages there is only one unified ProcessIDTable. Further, a given ProcessID only appears once in this table even if it is used in dozens of GUIDs. This allows the memory cost of representing ProcessIDs to be amortized over their many uses.

For instance, if on average in a message each ProcessID is used in 10 GUIDs, then the memory used per-GUID for ProcessIDs drops by a factor of 3—from 80 bits (if each GUID contained the full ProcessID) to 26 (with each GUID containing a 16-bit index of a 96-bit ProcessIDTable entry). When ProcessIDs grow to 176 bits under IPv6, this difference will be an even more dramatic reduction by a factor of 5.

$$5 = \frac{176}{16 + \frac{16+176}{10}}$$

The above estimates of savings are conservative, because in Object State messages, which are by far the most common kind of ISTP message, it is common for every GUID to come from the same process and therefore for every GUID to have the same ProcessID. In addition, GUIDs are one of the most common constituents of objects. Therefore, there might well be scores of GUIDs in an Object State message, all with the same ProcessID, leading to compression factors approaching 5 with current ProcessIDs and 11 with IPv6 ProcessIDs.

Allocating GUIDs. ISTP GUIDs are designed so that it is possible to use one indefinitely without having to worry about GUID collisions. However, it is pragmatically important not to do so. The compression ratios above depend critically on the assumption that almost all the GUIDs owned by a given process have the same ProcessID.

If, in the extreme, every object had a different ProcessID, then there would be no compression. (In fact, the use of indexes would cause a small expansion.)

If GUIDs were used permanently, then as the days wore on, the ratio of ProcessIDs to GUIDs in use would relentlessly rise toward 1.0 with a concomitant reduction in effective compression. Rather than let this happen, one should take the opportunity to remove old objects and create new ones with new GUIDs using currently active ProcessIDs whenever possible.

(Note that beacons are the recommended method for making a persistent mark. Because they use URLs, they provide an unbounded number of tags. These tags take up a significant amount of space. However, this space is only used when a truely persistent tag is actually needed.)

An example GUID implementation. ISTP does not specify how the ProcessID part of a GUID should be generated. However, one plausible way is to generate 80 bit ProcessIDs by concatenation Internet addresses, port numbers, and a generation counter.

InternetAddress 32 bits - machine process is on. (128 bits under IPv6.)
PortNumber 16 bits - identifies process on machine.
GenerationCounter 32 bits - unique each time a given process starts.

The InternetAddress is the address of the machine an ISTP process is running on. This allows the uniqueness of ISTP GUIDs to follow from the uniqueness already guaranteed by the DNS name service.

The PortNumber is an I/O port number. Whenever an ISTP process starts up, it attaches itself to a port. This port is used to differentiate between multiple processes on a machine. In particular, including this port number in the GUIDs guarantees that two currently running processes cannot have the same ProcessID.

The GenerationCounter consists of 32 bits that have a high probability of being different every time an ISTP process starts on a given machine. As an initial approximation, one might use time in seconds for this. However, something that also involves file system interaction and/or communication with a trusted server would be better, because it is conceivable that a process could be started, stopped and then restarted in less than a second. In addition, machine clocks can stop and be set backward.)

If the generation counter is a time in seconds, it can be incremented once per second without risking accidental GUID collision when a process restarts. This allows a process to create 2^{16} new objects per second.

13 Connection Status

Once a 1-1 connection exists between a process P and a server S, Connection Status messages are sent periodically by both P and S. (Since an ISTP process can be both a client and server at the same time, P may be a client for some 1-1 connections while it is a server for others. For a given connection, the client is the process that initiated the connection.)

Connection Status messages support a control dialog about the connection between P and S indicating that the connection continues to be operational and to provide the basis for estimating time differences between machines. They have the fields shown in Figure 11.

The MessageType and SendTime fields are discussed in Section 12. The TopicID is ignored. A logical value for it is therefore zero.

In a Communication Status message from S, the ProcessIDTable is empty. In a message from P, the ProcessIDTable specifies a list of (usually just one) ProcessID. The Connection Status message specifies (among other things) that the sending process P is utilizing each of these ProcessIDs as part of Owner IDs (see Section 14.2) and communication IDs (see Section 15). That is to say, every object owned by P has an Owner ID whose ProcessID is in the specified list and every GUID generated to identify a request for Locale-Based Communication has a ProcessID is in the list.

S records the ProcessIDs associated with each 1-1 connection to it and uses this information to determine which connection to use when sending replies to particular events. This is possible,

MessageType 12 bits - see Figure 10, value 1 indicates Connection Status Message.
Length 20 bits - = 24, see Figure 10.
SendTime 32 bits - see Figure 10.
TopicID 32 bits - see Figure 10.
NumberOfProcessIDs 16 bits - G, see Figure 10.
ProcessIDTable $G * 96$ bits - see Figure 10.
MaxDelay 32 bits - (Suggested) MaxDelay time in milliseconds.
Status 16 bits - status code.
InterveningMessages 16 bits - number of intervening messages.
LastSendTime 32 bits - SendTime of last Connection Status message.
TimeDifference 32 bits - estimated time difference in milliseconds.

Figure 11: Format of a Connection Status message.

because all requests from P to Content- and Locale-Based Communication servers are identified by GUID and/or Owner IDs with one of P's ProcessIDs.

In a Communication Status message from P, the MaxDelay field suggests a value for the parameter MaxDelay (see Section 13.2). In such a message a MaxDelay of zero indicates that P has no suggestion to make. In a message from S, the MaxDelay field specifies the MaxDelay parameter value to use for the 1-1 connection. In both cases, MaxDelay must be less than 3.5 days. It is typically at most a few seconds.

The remaining fields in a Communication Status message are interpreted symmetrically. They have exactly the same meaning in a message from P as in a message from S.

The Status value specifies an action the recipient process should perform.

0 = KeepAlive - indicates 1-1 connection is operating normally.
1 = Initialize - indicates the recipient process should send the current status of all pertinent information being communicated over the 1-1 connection. This may trigger the initial transmission of information, or it may trigger the retransmission of information that was somehow lost.
2 = Close - indicates 1-1 connection should be terminated.

The InterveningMessages is a count of the messages transmitted over the 1-1 connection between the last and current Connection Status messages. In the first Connection Status message sent by a process over a given 1-1 connection, the InterveningMessages field is zero.

The LastSendTime specifies the SendTime of the previous Connection Status message sent over the 1-1 connection. In the first Connection Status message sent over the connection, the LastSendTime is equal to the SendTime.

The TimeDifference in specifies the sending processes estimate of the average total nominal difference (according to its own clock) between when the sender processes a given message N from the other end of the connection when (according to the clock of the process at the other end of the connection) N was sent, see Section 13.1. This is a signed integer in milliseconds. The value $2^{31} - 1$ is used to indicate that the sending process has no information on which to base an actual estimate.

Status values. 1-1 connections are initiated from an ISTP process connecting to a server. (The initiator may be an ISTP server, the contacted process must be.) If the server agrees to the establishment of a 1-1 connection, the first message the server sends over the connection is a Connection Status message with Status Initialize. This is followed by whatever messages the server deems appropriate to initialize the connection. (This depends on what the connection is for.)

When it gets a Connection Status message with Status Initialize, the initiating process responds by sending whatever messages it deems appropriate to initialize the connection. (This also depends on what the connection is for.)

If at any time, either process decides that the connection should be closed they send a Connection Status message with Status Close and then close their end of the connection. The process receiving such a message should immediately close its end of the connection. (A key purpose of Status Close is so that a process can terminate a connection without it looking like the process crashed. This is important, for instance, because a server might want to take special action if a process it is talking to crashes.)

If at any time, either process determines that something has gone wrong either within itself, or over the 1-1 connection that leaves the process in doubt as to whether it has full and complete information about what was transmitted, it can send a another Connection Status message with Status Initialize. The process receiving such a message should immediately resend sufficient information to reinitialize the other process with appropriate information. What this information is depends on what subprotocol is involved.

Once a connection is open, both processes send Connection Status messages with Status KeepAlive whenever MaxDelay milliseconds pass without any other messages being sent. In support of the reliability checking discussed below, processes send Connection Status messages occasionally (e.g., once every ten times MaxDelay) even if there is ample other message traffic on the link.

Both processes continually monitor each 1-1 connection they are involved in. If significantly more than MaxDelay milliseconds (e.g., more than twice MaxDelay) pass without any messages being received, then a process concludes that the connection is broken. The process discovering the problem can take one of several actions when discovering this problem. It can send a Connection Status message with Status Initialize to try to get things going again. It can close the connection and try to open a new one. It can simply close the connection and report failure to the the part of the process that is using ISTP.

Reliability. The reliability of ISTP rests on the reliability of the 1-1 connections it uses. Since these are TCP connections, they are supposed to guarantee in-order non-duplicated delivery of every message sent. However, due to overflow of buffers and the like it is always possible for something to go wrong.

The LastSendTime and InterveningMessages fields are included in Connection Status messages so that an implementation of ISTP has a positive way to test that every message sent over a 1-1 connection is being properly received.

Whenever a process receives a Connection Status message, it compares the values of LastSendTime and InterveningMessages with locally maintained information about traffic on the connection. If it detects that a message has been lost, it immediately takes action as outlined above under the assumption that the connection has been broken.

13.1 Estimating Time Differences

Each time an ISTP message that is not an HTTP message arrives, computing the difference between the local clock time and the SendTime in the message provides an estimate of the TimeDifference between the sending and receiving processes. As a result, a process can build up a reasonable estimate of the TimeDifference between itself and another process by computing some sort of moving average based on single-message observations. (As part of this, it is probably wise to discard values that are obvious outliers.)

Note that the TimeDifference between two processes P and Q is the sum of three factors C, F, S where: C is the difference in time settings between the clocks in the two processes; F is the time of flight of messages from one process to the other; and S is the amount of time consumed by message handling software and delays fielding interrupts at the two ends. (To make S small, message receipt should be handled in a separate thread.)

Let the TimeDifference between P and Q as estimated by Q be

$$T_Q = C_Q + F_Q + S_Q$$

Similarly, let the TimeDifference as estimated by P be:

$$T_P = C_P + F_P + S_P$$

A key thing to notice is that $C_Q = -C_P$. If the network situation is symmetric, $F_Q = F_P$. If the software situation is symmetric, $S_Q = S_P$.

Time difference estimates are used in two critical places in ISTP. Estimates of total round trip communication time are used as part of processing Object State Summary messages, see Section 22.9. This can be estimated by adding two unilateral TimeDifference estimates T_Q and T_P:

$$T_Q + T_P = F_Q + S_Q + F_P + S_P$$

Unilateral estimates are used when processing Object State messages (see Section 14) to correct absolute times in objects.

In a symmetrical situation, clock differences can be estimated by taking the difference of two TimeDifference estimates.

$$\frac{T_Q - T_P}{2} = C_Q$$

13.2 MaxDelay

A fundamental time parameter of ISTP is a value (in milliseconds) called MaxDelay. This parameter is on the order of one to several seconds and can have a different value for each 1-1 Connection link. As discussed in detail below, the value MaxDelay is used for several separate things. In principle, the different uses of MaxDelay could make use of different values; however, as also discussed below, it is reasonable to use similar values. Therefore, in the interest of simplicity, only one value is used for each 1-1 Connection link. The value to use for a given link is specified by the server end of the link in the Connection Status messages it sends.

MessageType 12 bits - see Figure 10, value 2 indicates Object State.
Length 20 bits - see Figure 10.
SendTime 32 bits - see Figure 10.
TopicID 32 bits - see Figure 10.
NumberOfProcessIDs 16 bits - G, see Figure 10.
ProcessIDTable $G * 96$ bits - see Figure 10.
NumberOfDescriptions 16 bits - number of descriptions D in body of message.
Descriptions - D object descriptions (varying length)

Figure 12: Format of an Object State message.

14 Object State

By far the most commonly used ISTP message is the Object State message. These messages are the only messages that are sent by UDP. They are also sent by TCP over 1-1 connections. Each Object State message has the fields shown in Figure 12 and contains the descriptions of one or more objects.

In an Object State message, the value of the TopicID depends on why the Object State message is being sent. If the message is sent as part of Locale-Based Communication, then the TopicID is the GUID that was initially generated when a given process P initially sent a requested to the relevant Locale-Based Communication server S that it begin join the communication group. This is true whether the Object State message is sent by P to S, by S to P, or by P to some other client process Q.

Alternatively, if an Object State message is sent for the purpose of Content-Based Communication, then the TopicID is typically the GUID of a beacon or spBeaconMonitor object. If the message is sent from a client to a Content-Based Communication server, then the GUID is the GUID of the first object described in the message—this object typically being a beacon. If the message is sent from the server to a client, then the GUID is the GUID of the spBeaconMonitor object that triggered the response.

The NumberOfDescriptions specifies the number of objects described in the body of the message. The Descriptions describe the current state of objects using either full or differential descriptions as discussed in the following subsections.

When Object State messages are sent via UDP, the entire message is required to fit into a single packet. That is to say, it must be less than the Maximum Transmission Unit (MTU). What the MTU is depends on the transmission medium. Currently MTU's vary widely from only a couple hundred bytes (for modems) to 1,500 bytes (for Ethernet) and beyond. Under IPv6, there will be a minimum MTU of 60 bytes.

To ensure that Object State messages can fit into single UDP packets, it is required that every object description must be small enough so that an Object State message containing just that description could fit into a single UDP packet. When Object State messages are sent via TCP, they can (and are often required) to contain many descriptions and be larger than a single packet.

14.1 Rejecting Overly Late Messages

A receiving process P_k keeps track of the send and receipt times of the Object State messages it receives by UDP from each other process P_j. If P_k receives a message M from P_j that has an earlier SendTime than some other message it has already received from P_j, then M has been received out of order. (If two messages have the same SendTime, they have been sent at the same time and their order does not matter to ISTP.)

> If a late-arriving Object State message M arrives more than MaxDelay milliseconds after any previously received message from the same source with a later SendTime, then M is discarded without processing.

As discussed in Part III, the rejection of overly late messages is essential in order to place an upper bound on the length of time information has to be retained by a process P_k in order to be able to deal correctly with out-of-order Object State messages. (The MaxDelay value for a given UDP communication channel is chosen by the associated Locale-Based Communication server.)

Consider the following example. The table below shows the arrival times at a process P_k of six Object State messages sent 10 milliseconds apart by process P_j.

SendTime	arrival time
10	110
20	2135
30	130
40	1155
50	150
60	160

For simplicity, assume that the clocks in P_j and P_k are exactly synchronized. In that case, the messages are typically taking 100 milliseconds to go from one process to the other, but the messages sent at times 20 and 40 are severely delayed and arrive out of order with respect to the other messages.

If MaxDelay = 2000 (2 seconds) then the message sent at 40 can still be used when it arrives. However, the message sent at 20 is discarded because it arrives 2005 milliseconds after the message sent at 30. If MaxDelay = 3000, both messages could be used. If MaxDelay = 1000 both messages would have to be rejected. Note in this last case, the message sent at 40 is reject even though it arrives less than 1 second after the message sent at 60, because it arrives more than 1 second after the message sent at 50.

Note that the assumption of clock synchronization in this example is purely used to simplify the discussion of the example. The rejection of overly delayed messages does not require any assumption or knowledge about the degree of synchronization between the clocks in various processes. It operates based on differences of arrival times and assumes nothing about SendTimes other than that they increase monotonically with actual time.

Since SendTimes are represented modulo one week, overly late message rejection rests on the assumption that no message will ever arrive more than 3.5 days late. This is a relatively safe assumption given that lateness is typically a matter of only seconds.

To perform overly late message rejection a process has to maintain a table of the arrival times of recently received Object State messages. The size of this table is limited by MaxDelay.

DescriptionLength 16 bits - total size of the shared data in bytes.
Counter 16 bits - incremented whenever the object changes.
Name 32 bits - compressed GUID for the object.
Class 32 bits - compressed GUID for the object's class.
Owner 32 bits - compressed GUID identifying the owning process.
Locale 32 bits - compressed GUID identifying the locale containing the object.
SharedBits 32 bits - representing logical values.
  IsRemoved (the low-order bit, bit 0) - if 1, object has been removed.
  ForceReliable (the next low-order bit, bit 1) - if 1, forces communication by TCP.
  InhibitReliable (bit 2) - if 1, inhibits Object State Summary messages.

Figure 13: Layout of a shared object.

In particular, there never needs to be more than one message in this table that was received more than MaxDelay milliseconds ago.

If a message N was received more than MaxDelay milliseconds ago, then it will force any message sent before N to be discarded on arrival. If there is some other message N' sent later than N that was also received more than MaxDelay milliseconds ago, then N can be dropped from the table, because any message discarding that is forced by N is also forced by N'.

14.2 Shared Objects

To understand object descriptions, one must first understand the following facts about objects in ISTP. It is expected that applications will use many kinds of objects. In particular, they are allowed to define new kinds of objects. However, only a very few types of objects matter to ISTP.

Every shared object (i.e. every object communicated by ISTP) is an instance of a subclass of the class sp. The class sp specifies that every shared object includes the fields shown in Figure 13, which are the foundation of object descriptions.

The DescriptionLength specifies the length in bytes of a full description of the shared data in the object. As discussed in Section 14.3, the DescriptionLength is limited to a 13-bit unsigned integer. A 13-bit description length allows objects 4k bytes long. Since full descriptions are limited to fitting into single UDP messages this length is sufficient.

The Counter is used as an identifier for the state of an object A. Every time any shared part of A is modified, A's Counter is incremented.

The Name of an object is a GUID (see Section 12.2) that is used to uniquely refer to it from fields of other objects and in various ISTP messages.

Each object has many fields in addition to the ones in Figure 13. The Class is the GUID of a machine manipulable spClass object (see Section 14.2.5), that describes what all of the fields of the object are. ISTP can manipulate arbitrary application-specific objects with reference to their spClass descriptions.

The Owner of an object is a GUID that is used to uniquely refer to the process that owns the object. The owning process is the only process that can modify the shared fields of an object. From the perspective of the discussion here, the only important aspect of Owner field is that it allows every process to determine which objects it does and does not own.

The Locale is the GUID of the spLocale object for the locale that contains the object. From the perspective of the discussion here, the key aspect of the Locale field is that it selects the channel that is used for communicating information about the object.

The SharedBits field is used to compactly represent various logical values. Only three of these values for sp objects are relevant to ISTP. The low-order bit (bit 0) specifies whether an object has been removed. This bit is set to zero when an object is created and changed to one when it is removed. Once the bit has changed to one it can never change back to zero.

The other two bits control the way an object is communicated via the Locale-Based Communication subprotocol. When one, bit 1 forces reliable communication of the object using TCP via the appropriate Locale-Based Communication server, see Section 14. When one, bit 2 prevents ISTP from using Object State Summary messages to guarantee the reliable communication of the state of the object, see Section 16. This saves resources in situations where reliability is not necessary. The default setting of both bits is zero.

Counter comparisons. Counter incrementing and comparison is done using arithmetic modulo 2^{16} so that the largest positive counter value rolls over to the smallest. To accommodate this, all comparisons are done using modular arithmetic. That is to say the counter C is considered less than D (i.e., $C < D \bmod 2^{16}$) if $0 < D - C < 2^{15}$ or $D - C < -(2^{15})$. The counter value zero is reserved to mean a state in which nothing is known about the object A. The counter for an object starts at 1 and skips the value zero as it wraps around when counting up.

A 16-bit state counter allows ISTP to correctly order $(2^{16})/2 = 32,768$ object states. At a rate of 30 state changes per second for an object, this is 16 minutes worth of changes. This is plenty of time considering that ISTP's time horizon for object communication is on the order of seconds.

Ownership. A process P owns an object A if and only if the ProcessID of the Owner of A is a ProcessID associated with P. (If P has created more than 2^{16} objects, there will be more than one ProcessID associated with P.)

The ownership of an object can be changed by having the current owning process P_j change the Owner field to a different value. Following this, P_j will send out an Object State message specifying the new owner.

When a process P_j gives up ownership to another process P_k, P_j knows it is no longer the owner before P_k or any other process can find out that P_k is the owner. Therefore, there are brief periods when no process thinks that it is the owner of a given object. However, it can never happen that two processes think they are the owner of an object.

In the following sections, whenever it is said that "if process P owns an object A then ...", what this should be interpreted to mean is "if at a given moment a process P thinks that it owns an object A because the copy of A in P's world model has P as the owner, then ...". That is to say a process being the owner of an object is a strictly local property rather than being a global property supported by a centralized service.

14.2.1 spBeaconing

Beacon objects are the foundation of the Content-Based Communication subprotocol of ISTP, see Section 21. Beacon objects are instances of (subclasses of) the class spBeaconing, which specifies the shared fields shown in Figure 14 that are of relevance to ISTP. Except for

DescriptionLength 16 bits - see Figure 13.
Counter 16 bits - see Figure 13.
Name 32 bits - see Figure 13.
Class 32 bits - see Figure 13.
Owner 32 bits - see Figure 13.
Locale 32 bits - see Figure 13.
SharedBits 32 bits - see Figure 13.
    IsRemoved (bit 0) - see Figure 13.
    ForceReliable (bit 1) - see Figure 13.
    InhibitReliable (bit 2) - see Figure 13.
Tag 16 bits - offset of beacon tag string.

Figure 14: Layout of an spBeaconing object.

the last field, all these fields are inherited from the class sp.

The Tag is a byte offset (from the position of the field containing the offset) of a null terminated ASCII string. An offset is used so that strings of different lengths can be accommodated without wasting space in a beacon when a Tag string happens to be short. When a given beacon object is initially created, a fixed amount of space is allocated for its Tag (based on the string that will be used as the Tag). ISTP assumes that once a beacon has been created, its Tag string will never change. One consequence of this is that the length of a full description of a beacon remains constant throughout the lifetime of the beacon.

As discussed in Section 21, the Tag of a beacon is similar to a URL. It specifies which Content-Based Communication server services the beacon and acts as an identifying label for the beacon.

In ISTP, all processes are capable of being both clients and servers. A process P discovers that it is being requested to be a Content-Based communication server because it is sent (or has) an spBeaconing object B whose Tag specifies that P serves B.

14.2.2 spBeaconMonitor

As shown in Figure 15, the class spBeaconMonitor is a subclass of sp that adds one additional field, the Pattern. As discussed in Section 21, the Pattern of an spBeaconMonitor is essentially a URL with wild card characters and specifies what beacon Tags are being sought and what Content-Based Communication server to contact.

14.2.3 spLinking

Link objects act as small identifiers that are used to alert a process to large pieces of data it should obtain. The link objects themselves are communicated using the Locale-Based Communication subprotocol of ISTP, see Section 22. The corresponding data is fetched as part of the Object State Transmission subprotocol of ISTP, see Section 20.

Link objects are instances of (subclasses of) the class spLinking. The class spLinking has the fields shown in Figure 16, most of which are inherited from sp.

The URL is the byte offset (from the position of the field containing the offset) of a null

DescriptionLength 16 bits - see Figure 13.
Counter 16 bits - see Figure 13.
Name 32 bits - see Figure 13.
Class 32 bits - see Figure 13.
Owner 32 bits - see Figure 13.
Locale 32 bits - see Figure 13.
SharedBits 32 bits - see Figure 13.
    IsRemoved (bit 0) - see Figure 13.
    ForceReliable (bit 1) - see Figure 13.
    InhibitReliable (bit 2) - see Figure 13.
Pattern 16 bits - offset of query pattern string.

Figure 15: Layout of an spBeaconMonitor object.

terminated ASCII string. Just as with beacon Tags, an offset is used so that strings of different lengths can be accommodated without wasting space in a link when a URL string happens to be short. Also like Tags, that the URL string for a link will never change during the lifetime of a given link.

As discussed in Section 20, the URL of a link is a URL that identifies a (potentially large) piece of data. It is used to fetch the data in question. ISTP makes not assumptions about the format of the data identified by the URL of a link.

The Checksum is a cyclic redundancy code (CRC) checksum for the data pointed to by the URL. If the checksum in a link object changes, then the associated data is refetched.

Fixed positions of fields. Since the objects communicated by ISTP are related by multiple inheritance, it is not always possible for the fields inherited by class D from class C, to be at the

DescriptionLength 16 bits - see Figure 13.
Counter 16 bits - see Figure 13.
Name 32 bits - see Figure 13.
Class 32 bits - see Figure 13.
Owner 32 bits - see Figure 13.
Locale 32 bits - see Figure 13.
SharedBits 32 bits - see Figure 13.
    IsRemoved (bit 0) - see Figure 13.
    ForceReliable (bit 1) - see Figure 13.
    InhibitReliable (bit 2) - see Figure 13.
ignored 16 bits - data not relevant to ISTP.
URL 16 bits - offset of URL string.
Checksum 32 bits - checksum of URL data.

Figure 16: Layout of an spLinking object.

DescriptionLength 16 bits - see Figure 13.
Counter 16 bits - see Figure 13.
Name 32 bits - see Figure 13.
Class 32 bits - see Figure 13.
Owner 32 bits - see Figure 13.
Locale 32 bits - see Figure 13.
SharedBits 32 bits - see Figure 13.
  IsRemoved (bit 0) - see Figure 13.
  ForceReliable (bit 1) - see Figure 13.
  InhibitReliable (bit 2) - see Figure 13.
ignored 16 bits - field not relevant to ISTP.
URL 16 bits - see Section 14.2.3.
Checksum 32 bits - see Section 14.2.3.

Figure 17: Layout of an spMultilinking object.

same positions in instances of D as they are in instances of C. Rather, field accessors typically have to index through a table of offsets in the class descriptor.

For example, if the URL in a link was at the same position as the Tag in a beacon, then it would not be possible for a locale (which inherits from both spLinking and spBeaconing) to have its URL in the same place as a simple link and its Tag in the same place as a simple beacon.

However, the above notwithstanding, it is possible for certain specific fields to be given fixed positions by choosing their positions so that they cannot conflict no matter how many classes a given class inherits from. This is done for the small number of fields that ISTP cares about so that it can manipulate these fields without reference to the relevant spClass objects.

For example, the gap in the layout of spLinking objects above is strategically placed to accommodate the Tag in a beacon. In a link description, this gap is not necessarily vacant, because a particular link class that does not also inherit from spBeaconing might have a 16-bit field ISTP does not care about that can be placed in that position. The only restriction is that a field ISTP does care about cannot be put in that position.

14.2.4 spMultilinking

As shown in Figure 17, the class spMultilinking is a subclass of spLink, which has the same fields. from the perspective of ISTP, the only difference is that the associated data is required to have the following standard format:

```
URL checksum
 <additional indented lines can be interpreted in any way by the ReadData method.>
URL checksum
 <additional indented lines can be interpreted in any way by the ReadData method.>
...
```

A multilink is an index of up to 256 alternative pieces of data. (The limit of 256 is imposed in order to keep multilink differential object descriptions compact.)

DescriptionLength 16 bits - see Figure 13.
Counter 16 bits - see Figure 13.
Name 32 bits - see Figure 13.
Class 32 bits - see Figure 13.
Owner 32 bits - see Figure 13.
Locale 32 bits - see Figure 13.
SharedBits 32 bits - see Figure 13.
  IsRemoved (bit 0) - see Figure 13.
  ForceReliable (bit 1) - see Figure 13.
  InhibitReliable (bit 2) - see Figure 13.
ignored 16 bits - field not relevant to ISTP.
URL 16 bits - see Section 14.2.3.
Checksum 32 bits - see Section 14.2.3.

Figure 18: Layout of an spClass object.

Each entry in a multilink begins with a line containing a URL and checksum, which are logically equivalent to the URL and checksum in a simple link.

When multilinks are processed, exactly one of the entries in the index is used to retrieve the associated data. (A non-shared field called Multipart records which entry was chosen as the basis of the in-memory representation of the link data on a given occasion.)

The internal structure of the top-level data for a multilink has to be specified in ISTP, so that the data caching mechanism can be applied to the data referred to by the individual entries as well as to the index structure itself. It is also needed so that ISTP can support rapid modification via multilink differential descriptions 14.6.

14.2.5 spClass

As shown in Figure 18, the class spClass is also a subclass of spLink that has exactly the same shared fields. The data corresponding to an spClass object specifies (among other things) the layout of the fields in the class being described. Object descriptions (see the following sections) are interpreted based on this data. Some classes are built in but most are defined by applications.

Note that an spClass object describes the layout of the data in a full description of the object in an ISTP message. This layout features network byte order and IEEE floating point numbers. The layout in memory may well be different in a given implementation of ISTP. In particular, since PCs do not use network byte order, one cannot just copy data form memory to a message. Rather, one must reformat it. Fortunately, if this is done correctly in registers, this is not much slower than merely copying the fields. To facilitate differential description encoding/decoding, all 4-byte fields are aligned on 4-byte boundaries.

14.3 Full Descriptions

Full object descriptions can be understood in isolation without reference to any other information about an object. They have the form shown in Figure 19.

DescriptionFormatCode 3 bits - which for full descriptions is equal to 0.
DescriptionLength 13 bits - Bytes in description and therefore shared data.
Counter 16 bits - counter value for object.
Name 32 bits - compressed GUID that is the name of the object.
Class 32 bits - compressed GUID for spClass of object.
Owner 32 bits - compressed GUID identifying the owning process.
Locale 32 bits - compressed GUID identifying locale containing object.
SharedBits 32 bits - representing logical values.
... long [] - Other data words in object.

Figure 19: Layout of a full object description.

Every description starts with 3 bits that specify what kind of description it is. The remaining fields are network standard copies of the shared data in the object. The fact that the first 3 bits of the DescriptionLength are stolen to become a code specifying the type of description effectively limits the maximum Description length to only 2^{13}. The fields shown in Figure 19 are the ones that are required to be in any shared object, see Figure 13.

Full descriptions are straightforward to construct by copying the shared data from the object in question. They are equally straightforward to interpret by copying in the reverse direction. However, it must be realized that a certain amount of translation has to be applied to individual fields.

If the host machine uses network byte order and IEEE floating point internally, then it may be possible to layout an object in memory in almost exactly the way that it should appear in a full description. If so, then most of the effort of constructing a message boils down to merely copying blocks of memory. However, it a different byte order is used, then all non-string fields have to be converted to the proper byte order. In any event, special processing has to be applied to GUIDs and times in an object.

Encoding/Decoding GUIDs. As discussed above (Section 12.2), GUIDs are represented in messages in compressed form. If they are not compressed in memory, then they have to be compressed. Either way, an appropriate ProcessIDTable has to be constructed for the message containing the description. When a description is received, the compressed GUIDs have to be decoded with reference to the ProcessIDTable in the message.

Encoding Times. Absolute times in a description are represented in the time frame of the server associated with the communication channel used. If a description is sent over a 1-1 connection, then times are represented from the point of view of the server end of the connection. This means that time encode/decoding is trivial at the server end. The process at the other end encodes and decodes times based on its estimate of the time difference across the connection see Section 13.1.

If a description is sent via UDP, times are represented from the point of view of the associated Locale-Based Communication server. The time adjustment to use is computed in conjunction with the 1-1 connection to the server. (Note that Locale-Based Communication servers never send UDP messages to each other.)

Note that this approach means that client processes do not have to maintain estimates of

DescriptionFormatCode 3 bits - which for differential descriptions is 1.
BaseCounterDelta 5 bits - Delta from reference object state.
FirstCode 8 bits - First byte describing where changes have occurred.
Counter 16 bits - counter value for object.
Name 32 bits - compressed GUID that is the name of the object.
OtherCodes byte [] - Remaining bytes indicating positions of changes. (In groups of 4 to preserve alignment.)
Data long [] - new word data representing changes.

Figure 20: Layout of a differential description.

time difference between each other. Rather, each process estimates the time differences between itself and the various servers it is connected to.

A complexity of this approach is that changing locale an object is in may change the encoding of times because the server may change. However, the overhead of this is not all that great, because changing locales already requires sending full description of the object in the new locale.

14.4 Differential Descriptions

Differential descriptions describe changes in objects from one state to another. Their format is shown in Figure 20.

The Counter and Name specify which state of which object is being described. The BaseCounterDelta specifies which prior states the differential description can be decoded with respect to. The FirstCode and OtherCodes specify what parts of the object have changed. The Data specifies new values for these parts.

These changes are changes in the network representation of the object, not its in-memory representation on any one machine. That is to say, they specify incremental changes in the full description of an object. To facilitate differential description encoding/decoding, all 4-byte fields are aligned on 4-byte boundaries.

Full descriptions have the advantage that they can be decoded by themselves without reference to any other information about the object described. Full descriptions must be used any time the recipient may not have any prior knowledge of the object in question.

Differential descriptions have the advantage that they are much smaller than full descriptions. Experimentation suggests that they are smaller by a factor of 10 or more in many common situations. Therefore in the interest of saving bandwidth, differential descriptions should be used as often as possible.

Differential descriptions complicate the handling of byte order changes, GUIDs, and times in descriptions, because it is harder to figure out what part of a differential description is what. However, the bandwidth savings are well worth the extra complexity.

How the reference states are specified. The BaseCounterDelta is a blend of a linearly and exponentially represented number. If i is the integer corresponding to the 5 bits of BaseCounterDelta, then the number Δ below is the value represented by BaseCounterDelta.

$$\Delta = \max(i+1, \lfloor 2^{i-16} \rfloor)$$

Specifically, the possible values of Δ are:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
32, 64, 128, 256, 512, 1,024, 2,048, 4,096, 8,192, 16,384, and 32,768.

Rapid decoding and encoding can be done using a table of these numbers.

If BaseCounterDelta represents the number Δ, then the description can be interpreted based on any of the last Δ object states. The advantage of using a differential description with $\Delta > 1$ is that the description is robust in the face of the loss of some descriptions from a sequence. The cost of this is that a description with $\Delta > 1$ may have to contain more data than one that can only be decoded based on the single prior state. However, there are many common situations where this penalty is very low or even non-existent. For instance, if an object A is being moved around and changed many times, but these changes are confined solely to the X-Y-Z position of the object, then even when Δ is very large, the differential description of A still only has to describe how three words of data have changed.

The values chosen for BaseCounterDelta are somewhat arbitrary, but they have three key features. First, for small deltas, every number is represented. This gives high precision in that range. This is important, because Δ often has to be small and yet it is important for Δ to be as big as possible.

Second, for larger deltas, many numbers are skipped, but a very wide range is covered. This is important because it is often possible for Δ to be quite large. In that case, precision is not all that important, but it would be very unfortunate to limit Δ to some very small value. If BaseCounterDelta used a purely linear encoding, then the largest possible value would be only 32. That many changes could happen to an object in less than a second. (Additional bits could be added to BaseCounterDelta, however doing this would block the useful property that a 1-word change in an object can be expressed using a 3-word description, see below.)

Third, the largest number represented (32,768) effectively acts as infinity, because two counters compared modulo 2^{15} cannot differ by more than this amount. Therefore, a differential description with a BaseCounterDelta of 31 (which represents 32,768) can be decoded with respect to any previous description. This is particularly important because one key differential message always has this property. When an object is removed, then nothing matters about the object except for the bit that specifies that the object has been removed. Therefore a small differential description can be created that can be decoded relative to any prior state.

How changes are represented. The byte change Codes have the following form. Non-negative bytes indicate word offsets. The first offset is relative to the beginning of the entire object. After that, each offset is relative to the first word after the end of the prior change. Negative bytes (by their absolute values) indicate word run lengths. A byte of 127 signals the end of the change bytes. If there are two non-negative bytes in a row, the length associated with the first offset is one. The words of data are aligned in the description and copied into the object as specified by the byte Codes. For example,

```
(1 0, 80; 1203) (88088) (-3, 10, -1, 127) (A) (B) (C) (D)
```

specifies that state 1203 of object 880088 can be computed from state 1202 by

writing the word A at offset $80 \times 4 = 320$,
writing the word B at offset $81 \times 4 = 324$,

writing the word C at offset $82 \times 4 = 328$, and
writing the word D at offset $93 \times 4 = 372$.

This 28-byte message specifies a 16-byte change.

As a special case, if the very first byte is negative, it is still treated as an offset, the length is one and the set of change codes consists of just this one byte. This special case allows a one-word change to be specified in just 3 words. For example,

```
(1 2, -30; 1203) (88088) (A)
```

specifies that state 1203 of object 88088 can be computed from state 1200, 1201, or 1202 by writing the word A at offset $30 \times 4 = 120$. This 12-byte message specifies a 4-byte change.

If the first change is more than 126 words into an object, a dummy change has to be used on the way to the real change. Similarly, a change after a one word change can only move 126 words farther down the object. However, given that objects are required to be short enough so that a full description can fit into a single UDP packet, the 126-word maximum step size is not a significant problem.

Constructing differential descriptions. Constructing differential descriptions is harder than computing full descriptions, because one needs to know exactly which words in the object have changed. To compute descriptions where the BaseCounterDelta represents a delta of $\Delta = 1$, one needs to either directly know which words have changed (e.g., recorded in a bit mask, where ones indicate changed words) or have a record of the prior state so that comparison can reveal which words have changed. Given either of the above, constructing a differential description is somewhat laborious but straightforward.

To compute descriptions where $\Delta = 2$, one needs to know which words were altered due to either of the last two state changes. (Note that if a word was changed and then changed back to its old value, it still must be included in the description, in case the recipient has the last state rather than the state before last.) A straightforward way to support BaseCounterDeltas of $\Delta < N$ where N is relatively small is to save bit masks summarizing the N previous state changes. These can then be OR'ed together to yield a specification of which words to send for any $\Delta < N$. Each time an object is changed a new bit mask is computed and saved in a per-object queue, while the oldest bit mask (if there are more than N) is discarded.

For larger values of N, a reasonable approach is to compute how long a particular bit mask remains valid. In particular, on can maintain a bit mask that is the OR of all the bit masks above, along with a counter of how many states have elapsed. As long as the number of bits on in the mask remains low, the delta counter can be incremented and differential messages with large values of Δ can be constructed. If radical changes in the object ever cause a lot of bits to be turned on, then one can discard the bit mask and start again with a more limited mask that covers only a few prior states.

If the counter corresponding to a bit mask cannot be exactly represented by BaseCounterDelta, then one must use the next smaller number that can be represented. Using a larger number could lead to inaccurate decoding of the differential message.

14.5 Link Differential Descriptions

Sometimes, when the large data item pointed to by a link changes, only a very small portion of the data actually changes. In that case, the change can be rapidly communicated using a

DescriptionFormatCode 3 bits - which for link descriptions is 2.
BaseCounterDelta 5 bits - see Figure 20.
NumModifications 8 bits - number M of modifications
Counter 16 bits - see Figure 20.
Name 32 bits - see Figure 20.
NewChecksum 32 bits - new checksum value.
Modifications byte [] - M modification descriptions (varying length)

Figure 21: Layout of a link differential object description.

link differential description. If large amounts of the data change in arbitrary ways, then there is no way to communicate the change rapidly, because there is no way to avoid communicating large amounts of data. Like any description, link differential descriptions are required to fit into single UDP packets, because Object State Messages are required to do so.

A link differential description contains the fields shown in Figure 21.

The DescriptionFormatCode, BaseCounterDelta, Counter, and Name in a link differential description are the same as in an ordinary differential description except that the DescriptionFormatCode has a different value and the Name must identify a link object.

The NewChecksum specifies a new value for the Checksum in the link. This is the only part of the link object itself that can be modified by a link differential description. (An ordinary differential description would be used for changing the link itself.)

The NumModifications field species how many modifications are included in the description. Allowing only 255 modifications is not unreasonable, since the entire description is required to fit into a single UDP message.

The Modifications are a series of byte codes that specify how a secondary storage image of the data associated with the link should to be changed. As described in detail below, these bytes are editing instructions for modifying a cached value of the data into the desired new form.

It is possible that the in-memory representation of the data is radically different from its representation in secondary storage. (It could have been arbitrarily modified by the ReadData method.) In consequence, ISTP does not attempt to incrementally modify the in-memory representation of the data. Rather, ISTP incrementally modifies the data in secondary storage and leaves it up to the Link class in question to decide how the in-memory image should be changed.

Each spLink class has a method called ModifyData, which takes the NumModifications and Modifications as arguments and can incrementally modify the in-memory representation. The default definition of ModifyData ignores the Modifications and merely calls the ReadData method to recreate the in-memory representation from scratch.

Decoding the modifications. Link differential descriptions use a standard delete and insert style that is straightforward to decode and reasonably straightforward to encode as long as one does not worry overly much about getting an optimally compact encoding.

The modification bytes consist of one or more sets of editing instructions of the following form.

Skip 8 bits - a number of bytes to skip over unchanged, high order bit 0.
Delete 8 bits - a number of bytes to delete, high order bit 0
Insert 8 bits - a number B of bytes to insert, high order bit 0
Bytes byte [] - B bytes to be inserted.

The Skip, Delete, and Insert fields are limited to 7 active bits each. Any one of these bytes can be preceded by one or more bytes with the high order bit 1. Each such byte adds 7 more bits (as a prefix). This can be used to skip and/or delete and/or insert large numbers of bytes at once. The Bytes field specifies the new data to be inserted. Each subsequent editing instruction begins at the first byte after the previous one ended. (To fit into a single UDP message, a set of Modifications are limited to inserting in the neighborhood of 1000 new bytes.)

The first of the following three sets of editing instructions

```
8,3,5,t,h,e, ,b,
4,3,0,
5,4,7,i,c,a,t,i,o,n
```

specifies skipping 8 bytes, deleting 3 bytes and then inserting the 5 bytes specified. Together, the three sets of instructions would change

```
"This is a test of modifying."
```

into

```
"This is the best modification."
```

The following editing instruction (represented in binary)

```
10000100 10010010 01111000 0 10 A B
```

skips 10000100101111000 = 67,960 bytes, deletes none, and then inserts 2.

Decoding multi-byte codes like the above requires a certain amount of bit shifting, but the complexity is well worth it, because it allows small editing instructions to be represented very compactly. In particular, replacing one bunch of bytes with another can often be done with only 3 bytes of overhead in addition to the new bytes.

In the above, nothing has to be done to deal with network byte order and the like because the link data is merely treated as a string. If the link data is really some binary format, then this has to be handled correctly by the ReadData function, if need be, in a machine specific way. If multiple binary formats are needed, this can be supported using multilinks.

14.6 Multilink Differential Descriptions

The specialized differential description shown in Figure 22 is used to modify the data pointed to by an entry in a multilink.

A multilink differential description is the same as a link differential description except that the DescriptionFormatCode has a different value, the Name must identify a multilink object, and it has two additional fields.

DescriptionFormatCode 3 bits - which for multilink descriptions is 3.
BaseCounterDelta 5 bits - see Figure 20.
NumModifications 8 bits - see Figure 21.
Counter 16 bits - see Figure 20.
Name 32 bits - see Figure 20.
NewChecksum 32 bits - see Figure 21.
NewEntryChecksum 32 bits - new checksum for Multipart.
Multipart 8 bits - entry in multilink.
Modifications byte [] - see Figure 21.

Figure 22: Layout of a multilink differential object description.

The NewEntryChecksum provides a new checksum for one of the entries in the multilink index. This is the only part of the index itself that gets changed. (An ordinary link differential message could be used to modify other parts of the data index.)

The Multipart identifies one of the entries in the data index associated with the multilink. The modifications, which are identical to those in an ordinary link differential description, are applied to the data pointed to by the selected entry.

If (and only if) the Multipart field in the link object in memory is equal to the Multipart value in the multilink differential description, the ModifyData method is called to modify the in-memory representation. Otherwise, it is assumed that the in-memory representation is independent of the entry that was modified.

14.7 When Messages Are Sent

As part of its description of the Locale-Based Communication subprotocol of ISTP, Section 22, discusses exactly when when Object State messages are sent. This is summarized in a simplified way below.

Whenever a an object A that is owned by a process P is changed, P sends out one or more Object Description messages describing this change. (As much as possible, several changed objects are put together into the same message.)

In general, these messages are sent by multicast UDP. However, the ForceReliable bit, or lack of multicast support for communication with P, will cause TCP communication with the Locale-Based Communication server to be used instead.

After sending out an Object State Message, P monitors Object State Summary messages from S to make sure that S finds out about the change. If not, P informs S by TCP.

After giving up ownership of an object A, P cannot send out any other messages about the object, except that P must remember the state of the object in which the ownership changed as long as necessary to ensure that the server S finds out about the change (or later states). Once that has happened, P's responsibility for A ceases. (While the above is going on, and after, the new owner can send out messages about A).

Object State messages are used to communicate between an ISTP process and a Content-Based Communication server, see Section 21. In that case, they are always sent by TCP.

MessageType 12 bits - see Figure 10, value 5 indicates Locale Com Status.
Length 20 bits - see Figure 10.
SendTime 32 bits - see Figure 10.
TopicID 32 bits - compressed GUID identifying connection.
NumberOfProcessIDs 16 bits - G, see Figure 10.
ProcessIDTable $G * 96$ bits - see Figure 10.
Locale 32 bits - compressed GUID identifying the locale for communication.
Status 16 bits - status code.
MulticastAddress 32+16 bits - Internet address and port number.
AudioAddress 32+16 bits - Internet address and port number.
UseTCP 1 bit - if not zero, specifies communication via TCP.

Figure 23: Format of a Locale Com Status message.

15 Locale Com Status

When a 1-1 Connection exists between a process P and a Locale-Based Communication server S (see Section 22), P and S send Locale Com Status messages back and forth, to regulate Locale-Based Communication. P sends Locale Com Status messages to request initiation or changes in Locale-Based Communication and to specify termination. S sends Locale Com Status messages that specify initiation, changes, or termination.

As shown in Figure 23, the MessageType, TopicID, SendTime, NumberOfProcessIDs, and ProcessIDTable are the same as in the other message types.

When a Process P decides to request the initiation of communication in a locale L (see Section 15.2) it generates a GUID that acts as a *communication ID*. This GUID is used as the TopicID of a Locale Com Status Message P sends to the relevant server in order to request the initiation of communication. This GUID is then used as the TopicID of every Locale Com Status, Object State, and Object State Summary message that is relevant to P's communication in L, no matter whether a message is sent by P or S.

The Locale field specifies the locale communication is being requested or granted for. It is the same in every Locale Com Status message that is relevant to P's communication in L, no matter whether a message is sent by P or S.

The MulticastAddress field specifies the address to use when sending and receiving multicast Object State Messages about changes in objects. (This is only meaningful in messages sent by S. It is ignored in messages sent by P.)

The AudioAddress field specifies the address to use when sending and receiving multicast audio stream packets. A different address is used for this so that a process can choose to ignore audio. (This is only meaningful in messages sent by S. It is ignored in messages sent by P.)

The Status field specifies an action the recipient process should perform.

1 = Initialize - Indicates the recipient process should send the current status of all pertinent information relevant to the Locale-Based Communication group specified by the TopicID. This may trigger the initial transmission of information, or it may trigger the retransmission of information that was somehow lost.
2 = Close - indicates communication in the should be terminated.
3 = WriteOnly - Indicates P requests joining a communication group, but only or outputting information.

The Status WriteOnly is used only in a message sent from P to S. It indicates that P wants to join a communication group, but only for outputting information about objects.

The Status Initialize requests the (re)transmission of the current state of every object in the locale. In a message sent from P to S, this asks S to communicate to P the current state of every object in the locale. This is necessary if, and only if, P wishes to start listening for information about objects in L. In a message sent from S to P, the Status initialize asks P to communicate to S the current state of every object in the locale that is owned by P.

The Status Close indicates the cessation of communication in the locale. In a message sent from P to S, this specifies that P has stopped sending and receiving information. In a message sent from S to P, Status Close tells P to stop sending and receiving information as soon as possible. (If P does not do so, the server may forcibly evict the process.)

The UseTCP bit indicates that communication should occur via TCP rather than UDP, see Section 22.7. In a message sent from P to S, this acts as a suggestion that using TCP would be advisable. In a message sent from S to P, the UseTCP bit tells P that S will use TCP for all communication from other processes in the group to P and that P should communicate all its object information directly to S via TCP.

Requests for and specifications of MaxDelay values are communicated between P and S via Connection Status messages (see Section 13) sent over the same 1-1 connection that is used for the Locale Com Status messages about a particular communication group.

15.1 spLocale

In Spline, spLocale objects have a number of important uses (see [2, 4]). However, only one of these uses—breaking the world model up into chunks that are communicated separately—is relevant to ISTP.

The class spLocale is a subclass of both spBeaconing and spLinking. As shown in Figure 24, It inherits the shared fields from both classes.

The Tag field of a locale L acts as an ordinary spBeaconing Tag and specifies the process P that acts as the Content-Based server for L. In addition, the Tag specifies that P serves as the Locale-Based Communication server for the objects in L.

Every shared object specifies (via its Locale field see Figure 13) what locale it is in. (The Locale field of an spLocale object points to the locale itself.)

Locale-Based Communication of information about an object is communicated under the control of the Locale-Based Communication server specified by the locale the object is in. An orphaned object that is not in any locale (because its locale field is zero) is not communicated to anybody.

DescriptionLength 16 bits - see Figure 13.
Counter 16 bits - see Figure 13.
Name 32 bits - see Figure 13.
Class 32 bits - see Figure 13.
Owner 32 bits - see Figure 13.
Locale 32 bits - see Figure 13.
SharedBits 32 bits - see Figure 13.
  IsRemoved (bit 0) - see Figure 13.
  ForceReliable (bit 1) - see Figure 13.
  InhibitReliable (bit 2) - see Figure 13.
Tag 16 bits - see Figure 14.
URL 16 bits - see Figure 16.
Checksum 32 bits - see Figure 16.

Figure 24: Layout of an spLocale object.

As in any spLinking object, the URL points to a block of data. This data specifies two key pieces of information about a locale L that are relevant to ISTP: *a suggestion of what multicast addresses to use* for objects in L and a list of the other locales that are *neighbors* of L.

The data pointed to by the URL in a locale object must contain one or more locale descriptions of the following form. Each description begins with a line of the form "`NAME=`*name*", where *name* is the identifying name of the locale.

This is followed by a line beginning "`TAG=`" that specifies the Tag of the locale.

This is optionally followed by a line beginning "`MULTICASTRANGE=`" that contains two Internet addresses separated by a space. If omitted, this range takes on a default value selected by the Locale-Based Communication server for the locale. In either case, the exact address to use is chosen at run time by the appropriate Locale-Based Communication server. The chosen address is communicated to the process communicating in the locale via a Locale Com Status message.

The above preamble lines can be followed by additional lines not relevant to ISTP. This is then followed by block of lines for each neighbor. The first line of the block begins with "`NEIGHBOR=`*reference* where *reference* specifies the URL of the neighboring locale. This is followed by other lines containing information not relevant to ISTP.

For example, the URL `http://www.BigRetailer.com/cyberstore/locales` might refer to:

```
NAME=MainRoom
TAG=//www.cybermall.com/BigBookstore
MULTICASTRANGE=225.0.0.0 225.0.0.255
...
NEIGHBOR=http://www.cybermall.com/tenant/locales#MainHall
...
NEIGHBOR=#BackRoom
...
NAME=BackRoom
TAG=//www.BigRetailer.com/BigBookstore-2
NEIGHBOR=#MainRoom
...
```

DescriptionLength 16 bits - see Figure 13.
Counter 16 bits - see Figure 13.
Name 32 bits - see Figure 13.
Class 32 bits - see Figure 13.
Owner 32 bits - see Figure 13.
Locale 32 bits - see Figure 13.
SharedBits 32 bits - see Figure 13.
- IsRemoved (bit 0) - see Figure 13.
- ForceReliable (bit 1) - see Figure 13.
- InhibitReliable (bit 2) - see Figure 13.
- IgnoreNearby (bit 3) - if 1, says process cares only about one locale.
- Audio (bit 4) - if 1, indicates that process is interested in streaming audio data.

Figure 25: Layout of an spObserving object.

Note the use of standard HTTP syntax when referring to individual locale entries. For example, the data pointed to by `http://www.cybermall.com/locales` assumedly contains a reference to `http://www.BigRetailer.com/cyberstore#MainRoom`. Note also that while a file of locales will typically contain many links to locales in the same or nearby files, it must contain at least one absolute link to a locale in a far away file in order to be connected to the rest of a large virtual world.

In ISTP, all processes are capable of being both clients and servers. A process P discovers that it is being requested to be a Locale-Based communication server because it is sent (or has) an spLocale object L whose Tag specifies that P serves L.

15.2 spObserving

The decision of when to communicate in a given Locale L and what kind of communication is desired depends on what objects have been created by the locale process P. For example, if P owns any objects in L, then P must output data in the communication group for L. If P owns an spObserving object in L, then P must input data from the communication group for L. Before discussing these communication rules in full detail, one must first consider what is specified by an spObserving object.

As shown in Figure 25, the fields of an spObserving object are primarily those of any shared object. The Locale field of an spObserving object specifies the primary spLocale in which communication is desired. Two additional bit fields indicate what kind of communication is desired.

When it is true, the IgnoreNearby bit specifies that P is interested only in the locale L containing the spObserving object. When it is false, the IgnoreNearby bit specifies that P wishes to receive information about objects in L and about objects in every neighbor of L.

When it is true, the Audio bit specifies that P is interested in receiving streaming audio information as well as object information. When it is false the Audio bit specifies that P does not want to receive audio information. This means it will ignore any AudioAddress it finds out about.

MessageType 12 bits - see Figure 10, value 3 indicates Object State Summary.
Length 20 bits - see Figure 10.
SendTime 32 bits - see Figure 10.
TopicID 32 bits - see Figure 10.
NumberOfProcessIDs 16 bits - G, see Figure 10.
ProcessIDTable $G * 96$ bits - see Figure 10.
TableSize 16 bits - number entries in object table.
NumberOfFullEntries 16 bits - number F of new full entries.
NumberOfDifferentialEntries 16 bits - Number D of differential entries.
FullEntries - F full object summaries (64 bits each)
    ObjectsTableIndex 16-bits - specifies table position for object.
    CounterValue 16-bits - CounterValue for object.
    CompressedGUID 32-bits - identifies new object.
DifferentialEntries byte [] - D object change summaries.

Figure 26: Format of an Object State Summary message.

Deciding when to Communicate. For a given locale L, P needs to write in L if it owns any object in L. P needs to read in L if it owns an spObserving in L, or an spObserving that is in a neighbor of L and has the IgnoreNearby bit off.

The moment P starts to need to read or write in L it should request the initiation of communication by sending a Locale Com Status message to the appropriate server S. It should keep communication open as long as there is any reason to continue. (There may be several reasons at any one time and the set of reasons may be continually changing; however, as long as there is any reason communication should continue.)

Whenever P switches from not needing to read in L to needing to read, then it needs to send a Locale Com Status message with the Initialize bit on.

When P notes that it no longer needs to read or write, it should wrap up communication (see Section 22.5) and then send a Locale Com Status with Status Close.

An ISTP process needs to do streaming audio output in a locale L whenever it owns an spAudioSource object in L through which an audio stream is being sent. An ISTP process needs to do streaming audio input in a locale L whenever it owns an spObserving object in L with the Audio bit on, or an spObserving object in a neighbor of L with the Audio bit on and the IgnoreNeighbors bit off. Starting and stopping audio input and output merely requires starting and stopping streaming audio communication using the appropriate AudioAddress.

16 Object State Summary

Object State Summary messages are used to communicate information from a process P to another process Q about what states of what objects are known to P. An Object State Summary message contains the information shown in Figure 26. The first five fields have standard values as discussed above.

Object State Summary messages are used in two situations by the Locale-Based Communication protocol, see Section 22. In both situations the are sent over a 1-1 Connection link between a process P and a locale-based communication server S. In either case, the TopicID is the GUID of the identifying GUID generated P when it established the relevant connection to S, see Section 22.4.

Object State Summary messages are essentially differential messages that specify information in an *objects table*, see below. For each locale, the relevant server maintains a table of what objects are in the locale and the most recent counter value for each object. The primary use of Object State Summary messages is to communicate information about changes in this table from servers to their clients.

The TableSize specifies how many entries there are in the objects table. If the table in the receiver has a smaller table it is increased to the specified size. If the receiver has a larger table, the excess entries are discarded.

The NumberOfFullEntries specifies how many table entries are described in the FullEntries part of the message. Full entries have to be used whenever a new object is inserted in the table. The FullEntries field contains corresponding complete entry descriptions.

The NumberOfDifferentialEntries specifies how many table entries are described using compact differential descriptions. The DifferentialEntries field contains corresponding differential entry descriptions.

The objects table. Before discussing the payload of an Object State Summary message, it is necessary to discuss the objects table maintained in each process communicating via a given locale L. This table lists the (compressed) GUIDs of each object that exists in L and an associated counter value. In the server S for L, the counter value represents the most recent counter value for the object that is known to S. In a client process, the counter value is the value specified by the most recent Object State Summary message from S.

Each locale L has an associated objects table that is created by the server S for the locale. The server has total control of exactly what is in the table and is the only process that ever modifies the table. A client process P receives Object state Summary message from S that all P to create a copy of the objects table created by S. Based on this copy, P decides what communication repairs have to be made.

S constructs its master copy of the objects table incrementally by updating it every time it finds out new information about an object. The objects table in a client P is initially constructed as part of finding out what objects exist when P initially connects to L, see Section 22.4.

Object State Summary messages are used to keep the objects tables in each client P synchronized with the table in S and therefore to tell each P whether they have up-to-date information about all of the objects in L. Since a process P will typically be communicating in several locales, it will typically be linked to several servers and maintaining several objects tables.

Suppose that a locale L, contained 6 objects with the following GUIDs.

```
87659001369886598231597842876   87659001369886598231597836834
87659001369886598231597836782   87659001369886598231597836138
87659001369886598231597836788   10008863450083264046309916458
```

As noted above (section 12.2) using the ProcessIDTable

```
23 876590013698865982315978
88 100088634500832640463099
```

these GUIDs can be compactly represented as follows:

```
23 42876   23 36834
23 36782   23 36138
23 36788   88 16458
```

Using these compressed GUIDs, the object table created for L by S, would be something like the following—the last column specifying Counter values for each object.

```
23 42876   234
23 36834    45
23 36782   879
23 36138   294
23 36788    34
88 16458   528
```

Full entry descriptions. The FullEntries field contains triples of objects table indices, CounterValues and CompressedGUIDs. It specifies that an entry has the indicated data. The CounterValue 0 is used to indicate that an entry should be discarded from the table. This is done by marking the entry as unused by giving it a zero GUID and CounterValue. Because objects table indices are limited to 16 bits, there cannot be more than 65,536 objects in any one locale at any one time.

If the entry specified by a full entry description does not exist, it is created. Dynamic table expansion might be necessary. (Entries that have previously been discarded, see below, are reused as much as possible.)

The first of the following two full entry descriptions specifies that the table entry with index 2 (the third, since indexes are zero based) should have the CounterValue 193 and the (compressed) GUID 23 36834. The second description specifies that the entry with index 4 should be discarded. (What its GUID is/was does not matter.)

```
2,882,23 36834
4,0,0
```

The only time that full entries are required is when a new object enters a locale. Since this is an infrequent event in comparison to other changes, full entry descriptions need not be used often.

It is possible that an object could appear and then be removed in a single time interval, but this is very rare. In that situation, there would be a full entry specifying a counter value corresponding to the object having been removed.

In general, a full entry description will either refer to a table entry that is empty, or one that already has the specified GUID. However, no matter what, the full entry description supersedes whatever is in the table. This fact can be used by a server to make arbitrary changes in the objects table in a process—e.g., during a reinitialization.

Differential entry descriptions. The DifferentialEntries field is designed for maximum compactness. It uses a series of byte codes to specify changes in CounterValues for objects. The byte codes consist of one or more blocks of the following form.

Skip 8 bits - a number of entries to skip over unchanged, high order bit 0.
Increment 8 bits - amount to increment CounterValue, high order bit 0

The Skip and Increment fields are limited to 7 active bits each. Either of these bytes can be preceded by one or more additional bytes with the high order bit 1. Each such byte adds

7 more bits (as a prefix). This can be used to skip and/or increment by large amounts. Each subsequent skip begins at the first entry after the one modified by the previous block. The Increment value 0 is used to indicate that an entry should be discarded from the table. This is done by marking the entry as unused by giving it a zero GUID and CounterValue.

Assuming the six entry objects table used as an example above, the following two modification blocks specify the exact same changes as the full descriptions shown above. The first block skips 2 entries and then increments the CounterValue in the entry with index 2 by 3. The second block skips 1 more entry and then discards the next entry (the entry with index 4).

```
2,3
1,0
```

The following modification block (represented in binary)

```
10010010 01111000 10
```

skips 00100101111000 = 2,424 entries, and then increments the CounterValue in the 2,425th entry by 2.

The key advantage of differential entry descriptions is that they are compact—typically only 2 bytes rather than 8 bytes for a full description. In addition, because they do not contain any GUIDs, using differential entry descriptions can also reduce the size of the ProcessIDTable in an Object State Summary message.

For differential entry descriptions to be maximally compact, a large enough percentage of the objects in a table have to be changing that the changed objects are reasonably close together. In addition, the maximum counter difference has to be less than 128 between two Object State Summary messages. (Note that at 30 changes per second, more than 4 seconds are required to go through 128 changes.)

When maximally compact differential entry descriptions can be used and no full entry descriptions are required, Object State Summary messages have a header of 28 bytes (one GUIDPrefix is required for the TopicID) and can therefore describe N changed objects using just $28+2N$ bytes. If 100 objects are changing continually and Object State Summary messages are sent out once every second, this causes 1.8 kbps of traffic from S to each client process P.

In the interest of completeness, it is specified that full entry descriptions are processed in order and before differential entry descriptions. However, there is no reason why the same table entry would be altered twice by the same Object Sate Summary message.

Discarding table entries. Discarding a table entry is very different from removing an object. If an object A is removed, this is specified by a description in an Object State message specifying that A is removed. Subsequent to this removal, the server will eventually discard the relevant objects table entry, but it should wait a considerable time before doing so. In particular, it should wait long enough that every client process P has found out that A has been removed.

It is suggested that S wait a time like 10×MaxDelay after an object is removed before reusing its table entry. This should, with high probability, insure that each client P knows that the object has been removed before the table entry is reused. However, if some process does not get this information, the object will in any event eventually get removed due to the mechanism discussed for removing objects that have no objects table entry, see Section 22.8.

MessageType 12 bits - see Figure 10, value 4 indicates Multiple Object Remove.
Length 20 bits - see Figure 10.
SendTime 32 bits - see Figure 10.
TopicID 32 bits - see Figure 10.
NumberOfProcessIDs 16 bits - G, see Figure 10.
ProcessIDTable $G * 96$ bits - see Figure 10, indicates objects to remove.

Figure 27: Format of a Multiple Object Remove message.

Obtaining up to date information. As part of its description of the Locale-Based Communication subprotocol of ISTP, Section 22, discusses exactly when when Object State Summary messages are sent. This is summarized in a simplified way below.

Once every MaxDelay milliseconds, a Locale-Based Communication server S sends an Object State Summary message by TCP to each client process P. These tell P what P should know. If there is anything that P has failed to find out due to message loss, P sends an Object State Summary specifying what it does know, and S replies (by TCP) with up-to-date information about the objects in question.

By observing Object State Summary messages from S, P may conclude that S has failed to find out about some change made by P. If so, then P sends (by TCP) up-to-date information to S.

The InhibitReliable bit can be used to prevent the sending of Object State Summaries about a given object so that no resources are spent trying to make communication about the object reliable.

17 Multiple Object Remove

Multiple Object Remove messages are used to specify that all the objects owned by a particular process (or processes) are to be removed. It is typically used when a process crashes or otherwise disconnects from an ISTP session.

As shown in Figure 27, a Multiple Object Remove message contains just the 5 fields in any non-HTTP ISTP message (see Figure 10). However, these fields are interpreted in a different than they are most in other messages.

The TopicID is ignored. A logical value for it is therefore zero.

The ProcessIDTable specifies a list of ProcessIDs. The Multiple Object Remove message specifies that every object whose name is a GUID with one of the specified ProcessIDs should be removed. (The indexes in the ProcessIDTable are ignored.)

In situations where all the objects belonging to a process need to be removed, using a Multiple Object Remove message is much more compact than sending Object State messages specifying that each individual object should be removed. For instance, a process that owns a thousand objects, can remove them all with a single Multiple Object Remove message.

DescriptionLength 16 bits - see Figure 13.
Counter 16 bits - see Figure 13.
Name 32 bits - see Figure 13.
Class 32 bits - see Figure 13.
Owner 32 bits - see Figure 13.
Locale 32 bits - see Figure 13.
SharedBits 32 bits - see Figure 13.
  IsRemoved (bit 0) - see Figure 13.
  ForceReliable (bit 1) - see Figure 13.
  InhibitReliable (bit 2) - see Figure 13.

Figure 28: Layout of an spAudioSource object.

18 RTP

The Streaming Audio subprotocol of ISTP relies on the Real Time Protocol (RTP) and various standard audio encodings. It is the intention that if better streaming protocols become available then they will be used in addition to (or instead of) RTP. In any event, ISTP will continue to rely on standard message formats rather than developing its own proprietary formats. Since these formats are standard and loosely coupled to ISTP, they are not discussed further here.

18.1 spAudioSource

spAudioSource objects represent sources of sound. The layout of an spAudioSource object is shown in Figure 28. In Spline, these objects contain various pieces of data that control the way sound should be rendered; however, these are not shown in the figure because they are not relevant to ISTP. Nevertheless, spAudioSource objects are important from the perspective of ISTP, because the influence the way audio streams are communicated.

Like any other shared object, an spAudioSource is placed in a locale. This specifies the locale the corresponding sound is communicated in. In addition, the sound stream is tagged with the GUID of the corresponding spAudioSource so that a receiver can tell which sound stream is which. (As discussed in Section 15, sound is communicated using different multicast addresses than those used for the communication of shared objects.)

Part III
ISTP Subprotocols

This third part of the report describes the five subprotocols that comprise ISTP. It explains how the messages in Part II interact to support the subprotocols and discusses why the subprotocols are designed the way they are. Parts II and III are best read together, with Part II used as a reference when the various messages are discussed in Part III.

19 1-1 Connection

The 1-1 Connection subprotocol of ISTP supports reliable (TCP) communication between an ISTP process and an ISTP server process. It is used as part of the Content- and Locale-Based communication subprotocols. In general, a given process P will have several 1-1 Connections to several different servers. For a given server S, there may be several reasons why there is a 1-1 Connection from P to S. The link remains open as long as any of these reasons remains in force.

For each 1-1 Connection link, the processes at the ends of the link must maintain several pieces of information. This includes the MaxDelay value corresponding to the link. This value is chosen by the server end of the link and is communicated in a Connection Status message (Section 13). A second piece of information is a list of the ProcessIDs in the owner IDs used by the process at the other end of the Connection. This is specified in Connection Status messages. A third piece of information is an an estimate of the time delay between the two ends of the connection. This is calculated based on Connection Status messages as discussed in Section 13.1.

The reasons why a process might initiate or accept a 1-1 Connection are discussed in the sections on Content- and Locale-Based Communication. The steps that are performed by a process that wishes to initiate and maintain a 1-1 Connection with another process are shown in Figure 29. The steps that are performed by a process that wishes to accept and maintain a 1-1 Connection with another process are shown in Figure 30. Both algorithms are best understood as finite state automata.

In general, Connection Status messages are used as an internal part of the 1-1 Connection protocol and are not visible to the processes at the ends of a link. However, Connection Status messages with Status Initialize that appear in the middle of communication are sent and handled by these processes. In general, such a message should be sent when a process feels that it may have lost some data received over the link. Receiving such a message triggers a process to resend all the data necessary to recreate fully up-to-date information at the other end of the link. What is needed to do this depends on what the 1-1 Connection is being used for.

As discussed in conjunction with spBeaconing (Section 14.2.1) and spLocale (Section 15.1) objects, the addresses to use when connecting to ISTP servers are specified by spBeaconing Tags. To make these connections, it is essential to have a widely agreed port number to use when contacting a server. Given the enormous difficulty in getting a new such port number assigned, ISTP piggybacks on the number 80, which is already used by the World Wide Web. This is done by simply making beacon Tags be URLs.

Forwarding. There are a variety of reasons why the process that responds to an initial 1-1

"**Closed**" A process P that wishes to open a 1-1 Connection to a server S starts in this state which indicates that there is no connection.

(1) If P wishes to open a 1-1 Connection to S, open a two-way TCP connection to S.

(2) If the TCP connection is refused, remain in this state. (After a reasonable amount of time P can again attempt to connect.)

(3) If the TCP connection is accepted, send an appropriate HTTP Get message (Section 8) to S requesting a 1-1 Connection and go to state "Connecting".

"**Connecting**" When P is in this state, it means that P has requested the initiation of a 1-1 Connection to S, but this connection has not yet been established.

(1) If a Connection Status message (Section 13) with Status Initialize is received, go to state "Open".

(2) If an HTTP Redirected Reply message (Section 10) is received, send an HTTP Get to the specified machine and remain in this state.

(3) If an HTTP Not Found message (Section 11) or a Connection Status message with Status Close is received, the request for a connection is being refused. Close the TCP connection to S and go to state "closed". (P should wait a significantly long time before trying again to connect to S, if ever.)

(4) If any other message is received, this indicates that S is not properly following ISTP. Close the TCP connection to S and go to state "closed". (P should probably never again try to connect to S.)

"**Open**" When P is in this state, it means that a 1-1 Connection is open between P and S. When entering this state, report successful connection to system running on top of ISTP in process P. When leaving this state, report that the connection is broken.

(1) From time to time, process P will specify an ISTP message to send over the connection, forward it to S and remain in this state.

(2) Whenever MaxDelay milliseconds pass without any message being sent by P, or a significant amount of time (e.g., $10 \times$ MaxDelay) passes since the last Connection Status message was sent by P, P sends a Connection Status message with Status KeepAlive and remain in this state.

(3) If a Connection Status message with Status Close is received, close the TCP connection, and go to state "closed".

(4) If a Connection Status message with Status KeepAlive is received, process the data in it and stay in this state.

(5) If an HTTP message, a non-ISTP message, or a Connection Status message whose LastSendTime and InterveningMessages do not agree with past history is received, something is going wrong with the 1-1 Connection. Send a Connection Status message with Status Close, close the TCP connection, and go to state "closed".

(6) If any other non-HTTP ISTP message is received, forward the message to P as part of the data communicated over the 1-1 Connection and remain in this state.

Figure 29: What a process does to initiate and maintain a 1-1 Connection.

"**Closed**" A process S that wishes to receive 1-1 Connections from other processes P starts in this state which indicates that there is no connection.

(1) If a request is made to open a two-way TCP connection, and it is practical to open it, then do so.

(2) If a request is made to open a two-way TCP connection, and it is not practical to open it—e.g., if resources do not permit an additional connection—then do not open the connection and ignore the remaining steps below.

(3) If an HTTP Get message (Section 8) is received that requests a connection to S for an appropriate server function, then:

(A) If S chooses to allow the connection, reply with a Connection Status message (Section 13) with Status Initialize and go to state "Open".

(B) If S chooses to forward the request to a different machine, reply with an HTTP Redirected reply message (Section 10), close the TCP connection, and remain in state "closed".

(C) If S chooses not to allow the connection, reply with an HTTP Not Found message (Section 11), close the TCP connection, and remain in state "closed".

(4) If any other message is received, this indicates that P is not properly following ISTP. Ignore the message, close the TCP connection, and remain in state "closed".

"**Open**" When S is in this state, it means that a 1-1 Connection is open between P and S. When entering this state, report successful connection to S. When leaving this state, report that the connection is broken.

(1) From time to time, process S will specify an ISTP message to send over the connection, forward it to P and remain in this state.

(2) Whenever MaxDelay milliseconds pass without any message being sent, or a significant amount of time (e.g., $10 \times$ MaxDelay) passes since the last Connection Status message was sent, send a Connection Status message with Status KeepAlive and remain in this state.

(3) If a Connection Status message with Status Close is received, close the TCP connection, and go to state "closed".

(4) If a Connection Status message with Status KeepAlive is received, process the data in it and stay in this state.

(5) If an HTTP message, a non-ISTP message, or a Connection Status message whose LastSendTime and InterveningMessages do not agree with past history is received, something is going wrong with the 1-1 Connection. Send a Connection Status message with Status Close, close the TCP connection, and go to state "closed".

(6) If any other non-HTTP ISTP message is received, forward the message to S as part of the data communicated over the 1-1 Connection and remain in this state.

Figure 30: What a process does to accept and maintain a 1-1 Connection.

Connection request might want to pass the request on to another process rather than establish the connection itself. One reason is that one might want to have both an ordinary web server and an ISTP server on the same machine. To allow port 80 to be used when connecting to either process, the web server needs to forward ISTP requests to the ISTP server. (To make this easy, ISTP is designed so that all the messages involved in forwarding are ordinary HTTP messages and therefore this part of ISTP can be supported by an ordinary web server.)

Forwarding is also useful if an application wants to have a centralized controller for its Locale-Based Communication servers that can manage multiple servers and dispatch requests between them. This central controller can do load balancing, restarting etc., within the ISTP framework.

Admission control. In ISTP, admission control is handled by allowing a process to interact with some Content- and Locale-Based Communication servers and not others. In particular, when a process P attempts to make a 1-1 Connection to a server S, S can choose to reject the connection. This is done on the basis of the Internet address of P. (This is known as part of the protocol for establishing the underlying TCP Connection.)

Acceptance and rejection can be done either based on "white-lists" of who is permitted to connect or "black-lists" of who is not allowed to connect. The creation of these lists is done by processes outside the scope of ISTP.

For example, one might interact with an application server that (for a fee) adds you to a white list consulted by an ISTP Content-Based Communication server. Being on this list might make it possible to access a particular set of beacons thereby making it possible to enter a particular application.

Security and authentication. One would like complete privacy and authentication of communication over 1-1 Connections so that outside parties can neither eavesdrop nor insert erroneous messages. The fact that point-to-point TCP is being used as the underlying communication mechanism provides a certain minimal level of security and authentication. To go beyond this, ISTP would have to be extended to include encryption of messages and/or digital signatures for authentication.

20 Object State Transmission

The Object State Transmission subprotocol of ISTP is used to communicate the current state of an object from one process to another. It is used by the other subprotocols whenever they want to communicate information about an object.

Most of what occurs in the Object State Transmission protocol is specified as part of the description of Object State messages (Section 14). The following discusses a number of higher level issues concerning the way Object State messages are processed.

20.1 Processing Object Descriptions

When an Object State message is received by a process P, it is broken up into individual object descriptions D. Suppose that a description D describes an object A. D is processed by P as follows.

The Counter value (Figure 13) associated with the state of A currently known to P (if any) is compared with the counter value in D. If the counter in D is less than or equal to the current counter value, D is ignored. This can occur, for example, when D is in a message that has arrived out of order or more than once.

If the counter value in D is greater than the current Counter value for A, or P does not know about A, then the information in D is used as follows. First if P already knows about A, it must check whether D was sent by the process that P thinks currently owns D. If D does not come from the right sender, then its processing must be delayed until a message from the right owner arrives that changes the owner to the sender of D.

(Ownership compatibility is determined by comparing the ProcessID in the TopicID of the message that contained D. Note that for this to work, all the Owner IDs and communication IDs for a process must be assigned with the same ProcessID. Considerable name space for GUIDs must be reserved for this in conjunction with the initial ProcessID used by a process. If the process runs long enough, it could eventually run out of name space for communication IDs. This would require it to change all its owner IDs and communication IDs—moving them into a new ProcessID space—before continuing on.)

If the owner is compatible, then full descriptions (Section 14.3) can always be immediately processed to update (or create or remove) the object in question.

If a description is differential (Section 14.4), then it can be immediately processed as long as the sender is compatible with the owner and it is interpretable relative to the state of A currently known to P. If not, then there must be some intervening description that has not yet arrived.

Descriptions that are not immediately interpretable can be saved on per-object queues for later use when missing intervening descriptions arrive.

Because of the use of differential descriptions that can be interpreted based on several prior states, it is often possible to act immediately using a description even if the previous description is in a message that was lost or delayed. This limits the damage due to lost and delayed messages without having to detect they are missing or resend them.

Once a description has been used, the relevant object description queue (if any) is examined to see whether there are any other descriptions that can now be used. (One can choose to simply discard differential descriptions that cannot be immediately used. This is simpler, but reduces the ability of a process to make use of out-of-order messages. As long as differential descriptions span several states, this might not be a problem.)

The details of the way an individual description is interpreted are discussed in Section 14.

20.2 Links

Suppose there is a group of asynchronous processes (possibly communicating over a network, possibly geographically separated, and possibly participating in a distributed virtual environment) that wish to interact in real time while sharing data in large files such as 3D models, recorded sounds, recorded video, etc. This situation features several conflicting goals.

Equality – ensuring that each process has copies of the data files that are the same as the copies being used by other processes.

Low bandwidth – minimize the communication bandwidth used.

Rapid interaction – maximize the speed of communication of data changes in order to achieve near real-time interaction.

It is not possible to always satisfy all three goals above. In particular, if a large multi-megabyte file is radically changed, there is no alternative to communicating it between processes and given the current Internet, this communication will take several seconds, making near real-time interaction impossible. However, by using caching and differential messages, one can satisfying the goals in a wide range of situations. The key idea is to separate the problem of knowing what data a process should have from actually getting the data itself, thereby taking maximal advantage of caching. Specifically, ISTP breaks communicating large data objects into three parts:

First, small objects called 'links' (objects that implement the class spLinking, Section 14.2.3) contain URLs that identify large pieces of data. Links can be rapidly communicated to inform processes of what they should know. A link object also contains a Checksum for the data pointed to by the URL.

Second, when a process receives a link (or learns that the Checksum has changed) it uses the URL and Checksum to determine whether it has the data already in local secondary storage (e.g., on disk or a CD-ROM). If it does not, it uses HTTP to fetch the required data, which it caches in secondary storage.

The Checksum is vital because it lets the process know whether it has the correct (e.g., most current) value for the data indicated by the URL without having to fetch the data. The cache contains triples of URLS, checksums, and data values.

Third, the last phase of receiving (updating) a link object is creating an in-memory image of the data. This is not part of ISTP, but is described here briefly for completeness. An in-memory image is created by applying an application-specified operation to the data corresponding to the link. There are no restrictions on what this operation does beyond a requirement that if the operation is called twice on identical data, it will create identical results. This assumption is important because it allows the operation to be called only once when there are multiple links that contain the same URL and checksum, sharing the results among all the links.

Several things should be noted about the above. The main virtue of the scheme is that processes are provided with positive indications whenever data changes and the data itself is only communicated when it changes. Therefore, unnecessary data transfers are avoided while ensuring that each process has the same version of each piece of data. Applications can define new kinds of links for whatever kind of data they desire. Multiple links to the same data are handled very efficiently with full sharing both in the data cache and memory.

In general, the format of the data referred to by a link is irrelevant to ISTP. The only assumption ISTP makes is that it is a sequence of bytes that can be obtained by an HTTP Get request. However, one specific kind of link (spMultilinking, Section 14.2.4) is specified that allows a link to act as an index for a set of data. An application-specified operation is used to select which piece of data (if any) is relevant at a given moment. There are no restrictions on what this operation does beyond a requirement that if the operation is called twice on identical data, it will return identical results.

Each entry in a multilink begins with a line containing a URL and Checksum. These are logically equivalent to the URL and Checksum in a simple link. In the following, the processing of spLinking and spMultilinking objects is described together, with spLinking viewed as a simplified form of spMultilinking where only one URL/Checksum pair can be specified.

The sequence of steps that are applied when a process P is notified of the existence of (or a change in) an spLinking object are shown in Figure 31. The responsibility of ISTP is to obtain a local copy of the indicated data.

Link differential descriptions (Sections 14.5 & 14.6) short circuit the processing in Figure 31

(1) If a process P is informed of the existence of a link L or of a change in L, then it performs the following steps.
(2) Use the selection operation specified for L's class to determine which (if any) of the pieces of data corresponding to L are of interest to the application. If none are of interest, take no further action.
(3) If there is cached data local to P that corresponds to the URL and checksum selected in step (2), then no further action needs to be taken.
(4) Use a standard HTTP Get message to retrieve the specified data. Follow Redirection messages if any are received. If the data cannot be fetched, report failure and skip the following.
(5) Check that the proper data has been received by calculating its checksum and comparing it with the checksum specified in the link. If the checksums do not match report failure and skip the following.
(6) Update the local cache by storing the fetched data, URL, and checksum. (The data needs to be cached for as long as the object L continues to exist in P's world model. It can be very profitable to cache it for longer periods of time, but this is not required.)

Figure 31: What a process does to obtain link data.

by simultaneously specifying how a link and its associated data have changed.

21 Content-Based Communication

spBeaconing and spBeaconMonitor objects are transmitted using Locale-Based communication just like any other shared object. In addition, they are subject to Content-Based communication as described below.

Just like the HTTP world, the Content-Based Communication world of beacons in ISTP is a seamless world where any process can communicate with any other process; however, the responsibility for this communication is divided up between an unbounded number of Content-Based Communication server processes. With respect to any particular beacon or spBeaconMonitor, communication occurs in a central-server style. However, this is scalable because each individual server is only responsible for a relatively small number of beacons.

The Tag of an spBeaconing object (Section 14.2.1) and the Pattern of an spBeaconMonitor (Section 14.2.2) is a URL. The DNS name part of this URL specifies what Content-Based Communication server handles the object. (This allows an application to control load balancing and to be sure that its beacons are stored on servers of its choosing.)

The remainder of the Tag of a beacon acts as a label that can be used as the basis for queries. The remainder of the Pattern of an spBeaconMonitor tags can contain asterisks which match zero or more characters, with the sole restriction that an asterisk cannot appear in the DNS or Port part of the Pattern.

For example, you might have beacons with the Tags `"//www.cybermall.com/bicycle/EBF"`, `"//www.cybermall.com/bicycle/RCW"`, and `"//www.cybermall.com/unicycle/RCW"`. In that case, an spBeaconMonitor with the Pattern `"//www.cybermall.com/bicycle/*"` retrieves the first two beacons while `"//www.cybermall.com/*RCW"` matches the last two tags. (When matching a Tag and a Pattern, things are simple as long as the pattern contains only one asterisk. However, if there are multiple asterisks, things become more complex and greater computation is required to decided which Tags match the Pattern.)

Once a process P obtains one or more beacons via an spBeaconMonitor, up-to-date information about these beacons is communicated to P by the relevant Content-Based Communication server whenever any beacon changes.

21.1 How a Process Becomes a Server

In ISTP, all processes are capable of being both clients and servers. A process S discovers that it is being requested to be a Content-Based Communication server when it has (or receives) an spBeaconing (or spBeaconMonitor) object B that specifies that S should serve B. (The way B is communicated to S is explained below. It is erroneous if S is not in a position to actually serve B.)

When a process S discovers that it should serve a beacon B, it retains in its memory information about B, the class of B, and the locale B is in. In response to spBeaconMonitor requests, S replies with information about the matching beacons that it knows about, see below.

A process can refuse to serve a beacon owned by another process, by simply removing the beacon from its world model. In addition, when a Content-Based Communication server process is initially contacted by another process, it can forward or refuse the connection.

21.2 Beacon processing

A process P participates in Content-Based communication by creating and modifying spBeaconing and spBeaconMonitor objects. Figure 32 shows what an ISTP process does when a beacon is created. Figure 33 shows what a Content-Based Communication server S does when hearing about a beacon. Figure 34 shows the on-going processing that is performed by S as a result of spBeaconMonitor objects.

An spBeaconMonitor is not just a one-time request for information. Rather it is best thought of as opening a reliable communication channel between a process P and one or more other processes that have created beacons P is interested in. Whenever any of these other processes changes a beacon, this information is sent to P via the appropriate Content-Based Communication server.

Restarting. If P receives a Connection Status message with Status Initialize from S, then it should resend full information about every beacon P owns that is served by S. Similarly, if S receives a Connection Status message with Status Initialize from P, then it should resend full information about all the beacons that match spBeaconMonitor requests owned by P. These Initialize messages are sent by a process if it feels that it has somehow gotten out of synchronization.

Locales are beacons. spLocale objects are special kinds of beacons. ISTP leverages off this by using Content-Based Communication to allow a Locale-Based Communication server to find out about the locales that are the neighbors of the locales it serves.

Joining the ISTP world. To enter the ISTP world, you do not have to ask anyone for permission. Rather, you merely start an ISTP process and you are in. However, to interact with anything other than content you create yourself, you must share some object with some other process.

(1) If an spBeaconing or spBeaconMonitor object B owned by P appears, then do the following steps.
(2) If a 1-1 Connection link to the server S for B is not already open, open one.
(3) Send a full description of B to S using an Object State message. If B is in a locale L, include L in the message. (L is also a beacon, but L may not be served by S.) If either B or L has a class C that is not built in, included C in the message. If C has a super class that is not built in, include it as well, etc.
(4) Maintain the 1-1 Connection link to S. If B or L or C ever changes in the future, send updated information to S.
(5) If B is removed or changes ownership, communicate this fact to S. If after this, P no longer owns any beacon served by S, close the 1-1 Connection to S.

Figure 32: What a process P creating/modifying a beacon (monitor) does.

(1) If an spBeaconing or spBeaconMonitor object B not owned by S appears (or changes) and B specifies that S serves B, perform the following steps.
(2) Check that a 1-1 Connection link has been established from the process P that owns B. (That is how B was communicated to S.)
(3) Save information about B, the locale L B is in, and the class of B (if it is not built in) for future reference.
(4) If B is a newly appeared spBeaconMonitor, reply to P over the relevant 1-1 Connection link with an Object State message containing full descriptions of every beacon known to S that matches B's Pattern. When sending this information, include the locale each beacon is in and the classes of the beacons and locales if they are not built-in classes. If possible, all the above should be sent in a single Object State message so that it will appear simultaneously in P's world model as a single event.

Figure 33: What a server S does when hearing about a beacon (monitor).

(1) Whenever a spBeaconing object B appears, changes, or is removed, do the following steps.
(2) For each spBeaconMonitor M whose Pattern matches B's Tag, do the following.
(3) Send information about the current state of B to the process P that owns M. If B is newly created and its class is not built in, the class should be communicated as well. If B is newly created or has changed locales, then the locale (and perhaps its class) have to be communicated as well. However, if a beacon merely changes in some other way, the class and locale do not have to be communicated because they are already known to P. In addition, a differential description can be used for B.

Figure 34: What a server S does in response to an existing spBeaconMonitor.

There are only two ways to initiate a connection to another process. The first way is to create an spBeaconMonitor. This will lead to your process finding out about beacon objects and locales other processes own. By connecting to some of these locales, your process can then enter fully into the ISTP world.

The second way is to create a beacon in a locale and advertise the Tag of the beacon. At some later time, some other process may create an spBeaconMonitor that fetches the beacon and then may begin interacting with you.

22 Locale-Based Communication

The goal of Locale-Based Communication in ISTP is to simultaneously achieve:

Scalability – To hundreds of thousands of simultaneous communicating processes.
Rapid Interaction – Maximize the speed of communication of object changes in order to achieve near real-time interaction.
Low Bandwidth – Minimize the communication bandwidth used.
Reliability – Guarantee that object changes are eventually (if perhaps sometimes slowly) successfully communicated between the processes.
Rapid Joining – Allow a new process to join a communication group and rapidly become up to date on what all the other processes know.

When considering how to achieve these goals, it is important to consider the following spectrum of the way that processes can interact. At one extreme, there can be hundreds of thousands of processes, but an individual process may not actually be interacting with any other process. At the other extreme, a single process could simultaneously interact with hundreds of thousands of other processes. In the current computational world, this extreme is not supportable.

ISTP is designed for situations where there is no limit on the total number of interacting processes, but each individual process interacts with only a modest number of other processes.

22.1 Locales

Under the assumption above, ISTP achieves scalability by breaking a virtual world into many chunks called *locales*. Each locale is associated with a server and a multicast communication group. It is expected that each group will use a different multicast address; however, a given server may serve several locales. A given process is expected to belong to more than one, but only a small number of communication groups. A given group is required to include only a modest number of processes (at most a hundred or so).

Specifically, spLocale objects (Section 15.1) divide the shared world model being communicated by ISTP into disjoint chunks. Every shared object (Section 14.2) is either in exactly one locale, or in no locale. Objects that are in no locale are not communicated. Objects that are in a locale L are subject to Locale-Based Communication only in that one locale. A process indicates its intention to output information in a locale by placing objects in the locale. A process indicates its intention to receive information in a locale by placing spObserving objects (Section 15.2) in (or near) the locale.

Given the assumption above, locales allow the problem of supporting the communication of vast numbers of users to be reduced to supporting communication in groups that have only a modest number of participants. The fact that there may be vast numbers of groups implies significant use of network bandwidth. However, the communication within an individual group is otherwise unaffected by the communication in other groups. The rest of this section focuses on how ISTP supports communication within a single group.

22.2 Speed and Reliability.

When considering how to achieve the goals at the beginning of Section 22, it is also important to consider the following spectrum of the way that objects can change. An approach must work acceptably at all points of this spectrum and should work particularly well at whatever points are most likely in a particular application.

At one extreme, some objects change very frequently (e.g., tens of times a second or more). In this situation, it matters very little to recover a particular lost message describing a change, because it is not liable to be possible to do so before the change is rendered obsolete. Rather, the focus should be on always being able to utilize the latest information as soon as it arrives. In addition, one should use as few resources as possible on (useless) repair attempts.

At the other extreme, some objects change very infrequently (e.g., only once every few minutes or hours). In this situation, the exact moment a change occurs may or may not matter, but the fact of the change certainly matters. It is very important that each individual change is communicated. It is also important that if information is lost about a particular change, this is detected long before the next change occurs. In this situation, some sort of positive acknowledgment scheme is needed to detect lost messages.

In the middle of the spectrum are objects that change at moderate speed (e.g., once every few seconds or so). Here repair is important and must be relatively timely. This is a particularly difficult part of the spectrum to support well. Fortunately, it is plausible that many applications make use of the two ends of the spectrum more than the middle.

It should be realized that a single object may change rapidly for a while and then change slowly or not at all for a while. Therefore, an approach cannot rely on knowing in advance which objects will exhibit which kind of behavior. Rather, it must adjust dynamically to whatever is happening.

Distributed database and shared memory technology. One way to approach the goals above is to use standard distributed database or shared memory technology. In these approaches, the paramount goal held up above all others is insuring that at any moment when two processes access a given shared object, the two processes will always obtain the same values. To satisfy this goal, locks must be used to prevent processes from accessing objects at the wrong time. For example, suppose that process P wants to modify an object A. To do this P must:

(1) Check that no other process has locked A, waiting if necessary until the lock is free,
(2) Lock A so that other processes in the group are prevented from accessing A,
(3) Send messages to all the other processes notifying them that the lock is set,
(4) Wait until return messages arrive from each other process acknowledging the lock. This may result in discovering that some other process took the lock first, in which case P must return to step (1).
(5) Make the desired change in A,
(6) Send messages to all other processes specifying the change,
(7) Wait until return messages arrive from each other process acknowledging the change,

(8) Remove the lock on A,
(9) Send messages to all other processes saying that the lock is removed.

The handshaking above wastes bandwidth and dramatically slows interaction. Setting and freeing the lock on A requires multiple messages to be sent between P and the other processes in the group. The back and forth communication greatly increases the latency interval between the time P decides to change A and the earliest time at which any other process can access the change. Each message must be sent completely reliably which further increases bandwidth usage and latency. Finally, the latency rises rapidly as the number of processes in a group rises. As a result, standard distributed database or shared memory approaches cannot be used for the near real-time interaction of more than a handful of processes.

Approximate equality. To achieve near real-time interaction between even a moderate number of processes, one must abandon the otherwise desirable requirement that when two processes access a given shared object, the two processes will always obtain the same values. Rather, one must dispense with inter-process locking and allow temporary disagreements between processes about the values associated with an object. In particular, when a process P modifies an object A, there will be a short period of time before another process Q finds out about this change and during that time the values obtained by P and Q when they access A will differ.

It is convenient to also assume that each object has an owning process and only that process can modify the object. This avoids writers/writers problems and means that there does not have to be any means of arbitrating between simultaneous changes. If an application wants to have several processes that can alter a given object, then the ownership of the object can be transferred from one process to another. Alternatively, a single process can be appointed as arbiter of change requests for the object and be the process that actually makes the changes based on these requests. This essentially mimics exactly what would have to be happening if multiple processes were to directly modify the object, because there would in that case have to be some arbitration method. The following assumes that at any given moment each object has only one process that can alter it.

Given a relaxed equality constraint, several approaches have been used to attempt to meet the goal above: central server systems, Distributed Interactive Simulation (DIS), and reliable multicast.

Central server approaches. Central server approaches have each process in a group communicate the changes it makes to a central server, which then notifies the other processes. This approach does a good job of keeping the information known by the processes as close to the same as possible. It also does a good job of allowing rapid joining, because a new process can receive a rapid download from the central server of everything it is supposed to know. In addition, by sending the messages to and from the server using a reliable protocol like TCP, the central server approach can easily guarantee reliable delivery of information. This in turn allows the use of differential messages describing changes rather than complete objects. This reduces bandwidth significantly.

However, the central server approach has two problems. First, interaction speed is significantly limited, because all messages have to go first to the central server and then to the other processes in the group. In comparison to sending messages directly from one process to another, this adds an additional message flight time and adds the time required for the server to interpret the incoming message, decide what to do with it, and generate an outgoing message. Second, bandwidth needs are increased somewhat due to the need to send messages to the central server as well as to the other processes in the group.

DIS. Systems conforming to the Distributed Interactive Simulation standard (DIS) send messages about object changes directly from one process to another using what is effectively multicast messages using the UDP protocol. (Actually, early DIS systems use broadcast in dedicated subnetworks with special bridging hardware/software to forward messages from one subnetwork to another, but this is essentially what multicast capable network routers do.) The key virtue of this approach is that it communicates information between process at the maximum possible speed. In addition, multicast uses significantly less bandwidth than multiple point to point connections. However, there is no guarantee of delivery of UDP messages. Therefore DIS does not guarantee that a change made by one process will ever be known by a given other process.

To counteract the reliability problem, DIS takes two actions. First, each message sent contains full information about an object so that it can always be understood even if previous messages about the object have been lost. Second, DIS systems send out frequent (e.g., typically once every 5 seconds) 'keep-alive' messages specifying the current state of each object. This means that lost information is typically repaired within 5-10 seconds. It also means that a new process will be informed of everything it needs to know in 5-10 seconds.

The above notwithstanding, DIS is still left with four significant problems. First, the fact that differential messages cannot be used wastes a lot of bandwidth, because even when only a small part of an object is changing, a description of the whole object is continually being sent.

Second, the keep-alive messages waste a lot of bandwidth, because when an object is not changing at all, repeated messages are still sent describing the whole object.

Third, while keep-alive messages cause eventual repair, they do not cause fast repair. Therefore, the processes in the group can get significantly out of synchronization in what they believe about the data they share and near real-time interaction is impaired.

Fourth, joining is not rapid, because it takes 5-10 seconds for a new process to learn what the other processes know.

A clever part of DIS is that there is no central server process at all, and no need for any process to figure out what information other processes have received. Rather, all processes just forge ahead with little knowledge of the others. When few messages are lost, things work extremely well, albeit at the cost of significant additional bandwidth. When a significant number of messages are lost, things continue to work out with no increase in bandwidth usage, albeit with a reduction in real-time interaction.

Reliable multicast. A final piece of related work is research on reliable multicast protocols. In that work, the primary goal is to achieve low bandwidth operation using multicast messages, but to incorporate handshaking that ensures reliability. There are two basic ways to do this: with positive acknowledgment messages (ACKs) or negative acknowledgment messages (NAKs).

In ACK-based approaches, each recipient sends explicit ACKs of the receipt of the messages sent to it. As in protocols such as TCP, this allows the sender to know exactly what has to be resent and to whom. However, the problem with this is what is referred to as an "ACK explosion".

Suppose that a process P is sending messages to n other processes. Each time P sends a message, n ACKs are generated. This uses significant bandwidth and causes P to receive n

messages that it has to deal with for each message it sends out. (Note that in the group as a whole, there are n times as many ACK messages as data carrying messages. As a result, the ACK messages soon come to dominate all communication as the group grows large.) If the ACKs are themselves sent by multicast, then all the processes have to deal with all the ACKs. If the ACKs are send directly from the various processes back to the sending processes, then this means that $O(n^2)$ 1-to-1 channels are open and the bandwidth needed for communicating ACKs is increased.

In NAK-based approaches, control messages are sent only when messages are lost. Specifically, when a process Q notices that it has failed to receive a message M from another process P, it sends a NAK requesting that the message be resent. The advantage of this approach is that when messages are received, bandwidth is not wasted sending ACKs. However, there are still significant problems.

First, the primary way for Q to tell that it has missed M is for it to receive a different message sent by P after M. In comparison to using ACKs, this delays the time at which the loss of M can be detected and therefore repaired. This problem is particularly severe if P does not send any message after M. In that case, Q might never notice that M was lost. To counteract this problem, some kind of message must be sent that specifies what messages a process should have received. A pure NAK-based approach is only possible when each process sends a steady stream of messages.

Secondly, as with ACKs, if NAKs are themselves sent by multicast, then all the processes have to receive all the NAKs. If the NAKs are send directly from the various processes back to the sending processes, then this means that $O(n^2)$ 1-to-1 channels are open and the bandwidth needed for communicating NAKs is increased. In either case, the n NAKs that converge on the sender when a message is entirely lost is referred to as a "NAK implosion". The existence of this traffic causes difficulty at the sender that can further impede communication beyond whatever problem caused the communication to fail in the first place.

From the perspective of the goals posed here, reliable multicast protocols have several key problems. First, most of these protocols do not even attempt to support near real-time interaction or rapid joining, focusing instead on reliability, and low bandwidth.

Second, many of these protocols expend significant resources ensuring reliability features such as order of arrival that are not useful for solving the goals posed here.

Third, if ACKs are used, this uses a significant amount of bandwidth, even when few messages are being lost. If a significant number of messages are being lost, then bandwidth usage goes up even further due to the need to resend messages that are lost. If NAKs are used, then bandwidth usage is much lower when things go well, but ramps up much more steeply as messages are lost, due to the need to begin sending many NAKs in addition to resending messages.

In both cases, the basic behavior of requiring more bandwidth when messages are being lost is unfortunate since bandwidth limitations are a prime reason why messages get lost. Particularly in NAK-based approaches, this can cause a negative spiral where the initial onset of problems causes more problems.

Fourth, and perhaps worst, pushing directly for reliability at the low level of multicast messages themselves does not strike at the heart of the problem posed above. For example, suppose that process P_1 changes object A at time T_1 and sends a message M describing this change. Suppose in a NAK-based approach that at some later time T_2, a process P_2 discovers

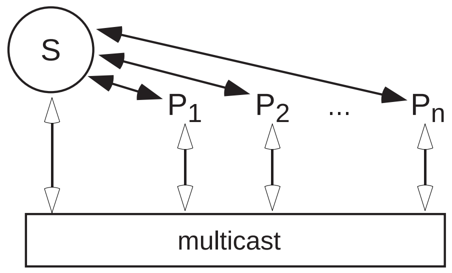

Figure 35: Locale-Based Communication Architecture.

that it has not received M. P_2 then sends a NAK requesting the retransmission of M. This is all well and good, but what P_2 really wants to get is not M, but what the state of A is at T_2. That is to say, the reliability that is desired is not necessarily the receipt of every message, but rather getting at all times the most up-to-date information about A possible.

The ISTP approach. The basic approach of ISTP for achieving rapid and yet reliable communication is to combine the best features of the central server and DIS approaches. ISTP uses a central server to support reliability and rapid joining, while using UDP multicast messaging to achieve rapid interaction. Because of the reliability provided, differential messages can be used to specify changes. This allows low bandwidth operation.

As shown in Figure 35, within a locale L, communication is controlled by a central Locale-Based Communication server S. The processes $P_1 \ldots P_n$ communicating in the locale communicate directly with each other (and with the server) via UDP multicast using an address chosen by the server. Each process P_j owns a number of objects. Whenever P_j changes one of the objects it owns, P_j multicasts a description of the change to the server and other processes.

Each process P_j has a 1-1 Connection to the server. This is used for the reliable communication of control information. In particular, the links are used for ACK messages from the central server and for messages making up for information that is lost during UDP communication.

A key feature of the ISTP approach is that there are no keep-alive messages about objects. This is valuable because keep-alive messages waste bandwidth without allowing timely repair or truly rapid joining.

In the interest of minimizing bandwidth, differential object descriptions are used wherever possible instead of full object descriptions. The simplest differential description specifies how to compute the new state of an object from the previous state. If this kind of description is used, a process cannot interpret a differential description D unless it knows the immediate previous state of the object. However, one can create differential descriptions that will compute the new state of an object from any of several previous states. If this is the case, then a description D can be interpreted even if the prior description was not received. Using this kind of multi-state differential descriptions allows ISTP to tolerate a certain number of lost messages without having to retransmit data.

When data has to be retransmitted, this is done by the central server on a per-object basis, rather than on a per-message basis. By using this approach, ISTP introduces reliability at a high level where it can take advantage of the constraints of the domain, rather than by using brute force at a low level. This saves bandwidth.

In particular, reliability is introduced to insure that processes end up with the latest state values for each object, not to insure that processes actually receive every low-level message. For instance, Object State messages that are lost, but soon rendered obsolete by subsequent Object State messages do not have to be (and therefore should not be) resent.

22.3 How a Process Becomes a Server

In ISTP, all processes are capable of being both clients and servers. A process S discovers that it is being requested to be a Locale-Based Communication Server when it has (or receives) an spLocale object L that specifies that S should serve L. The communication of L to S occurs by Content-Based Communication because locales are spBeaconing objects. (It is erroneous if S is not in a position to actually serve L.)

When a process S discovers that it should serve a locale L, it obtains the data corresponding to L (Section 15.1). This data specifies all the neighbors of L and what their Tags are. S uses spBeaconMonitor objects and Content-Based Communication to ensure that it will always have up-to-date information about every locale that neighbors L. (Note that things are often arranged so that S itself serves many of the neighbors of L.)

Once this is done, S selects multicast addresses to use for communication objects and sound in L, opens a connection to the addresses, and waits for processes to request joining into the communication group.

A process can refuse to serve a locale owned by another process, by simply removing the locale from its world model. In addition, when a Locale-Based Communication server process is initially contacted by another process, it can forward or refuse the connection.

22.4 Joining a Communication Group

A process P must output information via Locale-Based Communication in a locale L when it owns one or more objects in L. P must read information if it owns an spObserving object in L, or an spObserving that is in a neighbor of L and has the IgnoreNearby bit off.

When P transits from a state where it did not need to communication in L to one where it does, communication in L is initiated as outlined in Figure 36. The server contacted responds as outlined in Figure 37. Several things are worthy of note about this interaction.

From moment to moment, there may be many reasons why P wishes to communicate. It continues to do so whenever it has any reason. Whenever it goes from a state of not needing to read to one where it does need to read, it has to request a download of current object information by sending a Locale Com Status message with Status Initialize.

If S ever detects interference from other traffic on either of the addresses it has chosen, then it can pick new addresses and send new Locale Com Status messages specifying that the members of the group should change the addresses they are using. This channel-hopping approach can also be used to evict a rogue process from the group.

As soon as a Locale Com Status message permitting communication is received from the relevant server, a process can begin sending out information about the objects it owns and listening for information about other objects. However, it cannot understand incoming differential messages until it has gotten the current state downloaded from the server.

Joining a communication group is as fast as practically possible. In particular, the download happens by the fastest possible reliable means. (The complete time to join also includes connecting to the specified multicast address. Depending how the relevant routers work, this

(1) When P discovers that communication is necessary in a locale L, when it was not previously necessary, it performs the following steps.
(2) Generate a GUID G that will be used as an ID for communication done by P in L.
(3) If a 1-1 Connection link to the server S for L is not already open, open one.
(4) Send a Locale Com Status message C requesting that communication be initiated. In this message the TopicID is G, the Status is Initialize (if reading in L is desired) or WriteOnly, and the UseTCP bit is on if appropriate.
(5) If a Locale Com Status with Status Close is received from S, skip the following steps and give up the attempt to join the communication group at least for a while.
(6) If a Locale Com Status message M with a Status other than Close is received from S, open a connection to the specified MulticastAddress and begin receiving and/or sending object information. (If sending/receiving audio stream data is desired, open a connection to the specified AudioAddress as well.)
(7) If C has Status Initialize, Object State and Object State summary messages will be received from S. Process them to initialize the state of the world model and objects table for L.
(8) If at any time (including M) a Locale Com Status message with Status Initialize is received, then multicast an Object State message with full descriptions of all the objects owned by P in L.
(9) If at any time a Locale Com Status with Status Close is received, stop communication.

Figure 36: What a process P does to join a communication group.

(1) If a Locale Com Status message C for a locale L that a server S serves arrives from a process P with a Status other than Close, then save the TopicID G for use in subsequent messages and perform the following steps.
(2) Check that a 1-1 Connection link has been established from the process P associated with the TopicID of C. (That is how C was communicated to S.)
(3) If it is decided that P's request to begin communicating should not be granted, send a Locale Com Status message over the link to P with Status Close and skip the rest of these steps.
(4) Otherwise, if it is decided that the request should be granted, send a Locale Com Status message with Status Initialize telling P what multicast address to use and a Connection Status message specifying the MaxDelay to use, if that has not already been established for the 1-1 connection.
(5) If C has Status Initialize then send P an Object State message with full descriptions of the state of every object in L, every locale that is a neighbor of L, and any class that is needed in order to decode the descriptions of these neighbors. It is beneficial to do this in a single object state message so that all the relevant objects will appear together, but this is not necessary.
(6) If at any time a Locale Com Status with Status Close is received, stop communication.

Figure 37: What a server S does to allow a process to join a communication group.

can take a fair amount of time, but this is out of the control of ISTP.)

The initial download of information must contain descriptions of all the objects that have been removed or have left the locale less than MaxDelay seconds in the past, so that delayed messages about these objects will not cause them to erroneously appear in P.

After the initial download, S may discover additional information about neighbors of L. If so, this information is immediately communicated to P.

Once P is participating in the group, it must watch for new Locale Entry and Connection Status messages that might alter the MulticastAddress, AudioAddress, or MaxDelay value to use. If any of these parameters change, it must adjust its operation accordingly. If at any time, P receives a Locale Com Status with Status Initialize, it must resend complete information about the objects it owns in L.

A Connection Status message with Status Initialize is similar to a Locale Com Status message with Status Initialize except that the Connection Status message is more of a sledge hammer, forcing the communication of initialization information with regard to all communication relevant to a given 1-1 Connection. This may include information about several locales and/or beacons.

It is entirely possible that the system may desire to send messages via a locale before a connection with the locale has been fully established, because making the connection might take a second or more. To guarantee that things will work correctly, the system must construct the messages it wants to send and hold them in a queue until it can actually send them.

(It cannot just wait until the connection is made and then send current information, because some historical information can be essential. For example, suppose an object X is in a locale A that the current process P is connected to and P moves X to a locale B, P is not connected to, and them moves X again to a locale C before P can write any messages via B. Processes listening in A will be informed that X has gone to B, but will never be informed that X left B because P could not send this information, and has no currently existing reason to think it should, if connection is made to B only after X is in C.)

Admission control. As noted at the end of Section 19, admission control in ISTP is handled by allowing a process to interact with some Content- and Locale-Based Communication servers and not others. For example, one might interact with an application server that (for a fee) adds you to a white list consulted by an ISTP Locale-Based Communication server. Being on this list might make it possible to communicate in locales associated with a particular application.

22.5 Leaving a Communicating Group

A process P stops participating in Locale-Based Communication in a locale L when it no longer owns any objects in L and does not have any spObserving objects specifying that information should be obtained about objects in L.

As specified in Figure 38, P must continue the message repair part of the Locale-Based Communication protocol until it knows that the server S for L has adequate information about the current state of the objects (if any) that P previously had in L.

Once adequate information has been successfully transmitted, P tells S that communication has ceased. As indicated in Figure 39 there is little that S has to do when this happens.

Note that P can choose to stop reading in L while continuing to write (or vice versa) at any time without leaving the communication group. (Although not part of ISTP per se, it is likely

(1) When P discovers that communication is no longer necessary in a locale L, when it was previously necessary, it performs the following Steps.
(2) If P was reading multicast object data it stops doing so. In addition if P was reading or writing streaming audio data it stops doing so.
(3) After this, P must continue to monitor Object State Summary messages about L from S until it receives confirmation that S's knowledge of P's objects is sufficient to know that P no longer has any objects in L. (In order to reach this state, P may have to send some object information directly to S to insure that S gets information that was lost.)
(4) After P knows S has sufficient information, P sends a Locale Com Status message for L to S with Status Close.
(5) If there is now no reason for P to have a 1-1 Communication link open to S, P closes the link.

Figure 38: What a process P does to leave a communication group.

(1) If a Locale Com Status message for a locale L that a server S serves arrives from process P with Status Close, then discard any information relevant to P's communicating in L.

Figure 39: What a server S does when a process leaves a communication group.

that P will purge the objects in L that it does not own any time it stops reading information in L. Whether or not it does this, it must request reinitialization in order to come up to date about these objects before it can resume reading.)

Since actually disconnecting from L could take some time, it is possible for P to discover that it needs to reopen communication in L before the previous communication is completely closed down. Up until the moment when P has sent a Locale Com Status message to S with Status Close, P can merely start communication again without making any special notification to S. However, after that moment, P must reinitiate communication. In particular, it must go through all the steps for initiating communication (see Figure 36) including generating a new communication ID.

If P breaks its connection to S when S still thinks that P has objects in L, ISTP does not specify what should happen to these objects. The server could choose to maintain the existence of these objects in the hope that P will reconnect, but is allowed to remove them. (Note that in normal operation, this situation should only arise if P crashes.)

22.6 Communicating Object State

On a frequent basis (e.g., once every 30-100) milliseconds, a process P sends out one or more Object State messages describing all the objects it owns in L that have changed since the last time P sent messages. At a similar rate, it processes Object State messages sent by others.

Output flow control is essential to the smooth and efficient running of ISTP. If a processes sends out sudden large bursts of messages, this is almost guaranteed to cause problems clogging the network or overrunning buffers at one or more receivers. Instead of sending sudden bursts of traffic, an ISTP process should use a separate thread to spread its output traffic across time. To maintain real-time interaction, this spread should not be very great, but spreading across 10 milliseconds or so can make a big difference.

Messages are sent out using UDP multicast. Each message is sent in a single packet and each packet contains just one message. The address to use is specified by the server S in Locale Com Status Messages.

At a given moment, if no object has been modified, no message is sent. If several objects have changed, then as many descriptions as possible are packed into each message. However, each message must fit in a single UDP packet. (Grouping descriptions significantly decreases bandwidth usage and improves processing performance at the receivers.)

To minimize bandwidth, differential descriptions are used whenever possible. Full descriptions must be used whenever an object is first communicated to the group (i.e., when it is first created). After that, differential descriptions are possible. Whenever practical, differential messages are constructed so that they are not relative just to the last state of the object, but all the way back to the initial full message, or failing that, back at least several states.

Having differential descriptions interpretable based on the state before last is a huge advance in being able to tolerate lost messages over going back just one state, which requires the receipt of every message. Going back more than two states has advantages, but diminishing returns. Nevertheless, if one small part of an object is being changed rapidly, then one may be able to have differential descriptions interpretable across many states with no added costs.

If the data corresponding to a link object changes, then P can choose to send a differential link description or a differential multilink description instead of a simple object description. How P chooses to do this is outside the scope of ISTP. However, using these specialized messages when possible allows greater interaction speed.

Since UDP is not a reliable protocol, a given message M sent by P may arrive at another process Q: never, multiple times, and/or out of order with respect to other messages sent by P. Q must be able to deal with all these situations. This is done primarily on a per-object description basis based on the Counter values in object descriptions, rather than on a per-message basis. However, as noted in Section 14.1 a key feature of ISTP is that Q must reject all incoming messages that arrive more than MaxDelay milliseconds after some already received message that was sent at a later time.

If an Object State message is not discarded as being too late, then the object descriptions in it are processed individually as specified in Section 20

Note that the multicast Object State messages above are received not only by the other processes in the group, but also by the server S. Just like the various processes, S uses the messages to maintain a record of the current state of every object.

Removing objects. One issue that needs special discussion is what happens when a shared object is removed. When an object is being removed, one can use a differential description that can always be interpreted, because the only fact relevant about the object is that it is removed. Once an object has been removed, a potential problem could arise.

Suppose that all trace of a removed object A were removed from process Q. If so, then a subsequent out-of-order full description of A would appear to be a message specifying the creation of A, and would cause A to erroneously reappear in Q's memory. To avoid this, a record is maintained about the removal of A for MaxDelay milliseconds so that late arriving descriptions can be successfully ignored. If there were no lateness limit MaxDelay, then every object removed would have to be remembered forever by Q, in order for out-of-order description rejection to be supported.

To allow a process that has just entered a session to properly handle out-of-order messages about recently removed objects, the initial download of information about the objects in L must include information about all recently removed objects.

Changing an object's locale. In general, having multiple simultaneous communication groups does not present any fundamental complications. However, there is one key thing that must be addressed—what happens when an object moves from one locale to another.

First, whenever an object changes locales, a full message describing the object has to be sent in the new locale and a possibly differential message has to be sent in the old locale specifying that the object has left the locale.

Second, just as the removal of an object has to be remembered for MaxDelay time, the leaving of an object from a locale has to be remembered for MaxDelay time, so that out-of-order messages will not erroneously cause an object to reappear in the locale.

Third, just as the initial download of information about objects in a locale must include information about recently removed objects, it must contain information about objects that have recently left the locale.

Security and authentication. One would like complete privacy and authentication during the communication of object state so that outside parties can neither eavesdrop nor insert erroneous messages. The fact that dynamically chosen multicast address are being used for this provides a modicum of security and authentication, because processes that are not supposed to be communicating in a locale will have some difficulty figuring out what addresses to use.

To achieve greater security, ISTP would have to be extended to include encryption of messages and/or digital signatures for authentication. This could be done using hidden, rapidly changed keys chosen and distributed by Locale-Based Communication servers. (Security in the Locale-Based Communication protocol requires security in the 1-1 Connection protocol.)

The above only protects from outsiders, if you want to also protect from insiders who try to impersonate each other, you have to have per-sender keys and each receiver has to know about all senders and check that the messages received come from who they claim to come from.

22.7 Simulating Multicast

For simplicity, the discussion above assumed that all the processes in a group can be in multicast communication with each other. However, given the current state of the Internet, this may not be possible for many reasons including: many routers are not multicast capable and many firewalls will not allow multicast traffic to pass through. Therefore, ISTP includes the capability to do communication via simulated multicast using TCP rather than actual UDP multicast.

In the simulated multicast mode, a process P does all of its communication with the server S rather than directly with other processes. In particular, all the Object State messages it would have sent by UDP multicast, it instead sends directly to the server over its TCP connection to the server. Similarly, all the messages P would have received by multicast it receives over the TCP connection instead. (To facilitate this, everything is arranged in ISTP so that Object State messages can be correctly interpreted no matter what communication channel they arrive on.)

In simulated multicast mode, ISTP essentially operates in a central server mode and has no communication speed advantage over other central server designs. This mode is included in ISTP purely to allow graceful degradation when a given process is not capable of multicast communication with other processes in the group.

Note that given a group of processes, the situation involving multicast capabilities might be very complex featuring: multiple disconnected subgroups that are multicast capable within each subgroup, processes that can send multicast but not receive it (and vice versa) and dynamic changes where processes are capable of multicast communication at some moments but not others. ISTP does not attempt to optimally use multicast in all this situations, rather it attempts to work well in a few common situations while working correctly in all situations.

In particular, the communication group is divided into two parts: one (which must include the server) where every process can multicast send (and receive) to (and from) every other, and the remainder where TCP is used for all communication. Therefore, each process P is tagged as either using multicast communication or not. Automatic switching from multicast capable to not is supported, but there is no automatic support for the reverse.

Forcing TCP communication. If a process P receives a Locale Com Status message with the UseTCP bit on, it stops sending its relevant output via multicast and instead sends all of it directly to the server via TCP. If the UseTCP bit is on, then the value of the MulticastAddress field is irrelevant. P does not try to open a connection to the address and neither sends nor receives on it. (If a multicast connection was open, P closes it.)

Note that a Locale Com Status message with UseTCP on might arrive to initiate communication, or on the middle of communication. If P subsequently receives a Locale Com Status message with UseTCP off, then P will attempt to resume the use of multicast.

If the server S has told a process P not to use multicast, then S forwards all the information originating from other processes to P via TCP and takes the information from P and multicasts it to the processes in the group that are multicast capable.

The decision of whether a process P uses multicast is a joint one between P and the server S. It can be done unilaterally by either party by direct request. Specifically, S can tell P not to use multicast as described above. Similarly, P can request that multicast not be used.

If the server receives a Locale Com Status message from P that has the UseTCP bit on, then it should take this as a very strong request to reply with a Locale Com Status message of its own that also has the UseTCP bit on. A process P should turn this bit on initially if it has good reason to know that it is not multicast capable. Otherwise, there will be a period of low quality communication before ISTP automatically switches P to TCP mode (see below).

Note that such a Locale Com Status from P could be sent to initiate communication or in the middle of communication to trigger a change. It is possible for an Locale Com Status from P asking for TCP communication to be followed later by one asking that multicast communication be resumed.

Using the bits above, either P or S can specify the use of TCP from the moment that P joins the group. However, it is expected that one might often want to be more optimistic, initially trying multicast and only switching to TCP if the multicast fails. To do this, P and S start out with multicast and observe the error rate in communication.

If P observes (based on Object State Summary messages from S) that a low percentage of its multicast output is getting to S, then P should send a new Locale Com Status message requesting a switch to TCP.

If S observes (based on requests from P for object updates) that a low percentage of the data sent from other processes is reaching P, then S should send a new Locale Com Status message switching P into TCP mode. (For optimum performance, S should individually monitor how communication is going between each pair of processes in the group, but this is probably not necessary in most situations.)

Switching back to multicast. Using the above, it is easy to dynamically switch from multicast to TCP mode. However, once a process P is in TCP mode, there are no more attempts at multicast communication with P and therefore no basis on which to decide that one could switch successfully back to multicast mode. However, there are various approaches that could be used to make such a decision.

First, the server could occasionally switch P back into multicast mode and see if it worked. The price for this would be periods of reduced communication effectiveness. Therefore, if S takes this approach, it should reduce the frequency of its attempts if it meets with consistent failure.

Second, one could have P keep its multicast port open during TCP operation and create some experimental traffic specifically to assess whether multicast communication starts working. This is more complex, but allows the multicast connection to be assessed without forcing the application to suffer periods of poor communication.

Using one of the above approaches might be a good idea if multicast was working and unexpectedly stopped working. However, they are probably not worth the trouble if multicast never worked.

22.8 Object State Summaries

To inform the process P of what it should know, the server sends out periodic Object State Summary messages once each MaxDelay seconds. (The MaxDelay value used is the one associated with the 1-1 Connection to P and may be different for different processes being served by S.)

As discussed in Section 16, these Object State Summary messages are used to maintain *objects tables* in each process P in a communication group that specify what information each process should know about the objects being shared.

The objects table for a process P is initially constructed as part of the initial download of information from S when P joins a communication group. S constructs its master copy of the objects table incrementally by updating it every time it finds out new information about an object.

What P does when there is a change in the counter value C associated with an object A in P's copy of the objects table for a locale L depends on whether P does or does not own A.

Remotely owned objects. If P does not own A, then P checks to see whether it has up-to-date information about A. If P does not know about A at all, or has a smaller counter value for A, then P sends a request to S for updated information as described in the next section. Note P might have a larger value for the counter, because it might have received information from the owner of A that is not yet included in the summary from S. This is not a problem.

(For P to receive an Object State Summary message M with state C for A, the process that owns A must have sent out a message N with state C of A that S received and processed. P should have received N and been able to process it before receiving M under the assumption that sending a message from another process to S and then from S to P should take longer than sending a message directly from the other process to P.)

Locally owned objects. If P owns an object A, then the Object State Summary acts as a positive acknowledgment of the receipt of information sent by P to S. P must know a CounterValue greater than or equal to C (even if A has been removed).

If P has a larger CounterValue, then it might be the case that the latest message sent out about A got lost and therefore did not reach S. Alternatively, it might be the case that the message is proceeding on its way, but just did not get to S before the Object State Summary message was created. P has to decide which of these two situations is most likely. It can do this based on its estimate of the flight time of messages between itself and S, see Section 13.1. (Note that on the current internet, this flight time can be quite long.) If P concludes that a message was lost, then it resends the message as described in the next section. Note that P also has to consider whether a message has been lost if it owns an object that never gets into the objects table at all.

When P removes an object A, it must remember this fact however long is necessary to receive an acknowledgment that S knows that A has been removed. This will typically require P to remember that A has been removed much longer than it would need if all it were doing was rejecting out of order descriptions. (If P forgot about A before getting an acknowledge and the message specifying removal somehow failed to get to S, then a subsequent Object State Summary message could force A to erroneously reappear.)

Removing erroneous objects. A final way the objects table is used concerns objects that are in P's world model, but not in the table. Suppose object A exists, but has no table entry. This is a normal occurrence if A was just created (by P or another process) and the existence of A has not yet made it into an Object State Summary message. However, this situation should not last long.

If P owns A, then P will eventually send information about A to S, via TCP if necessary, and the problem will be resolved.

If P does not own A and yet A remains absent from the objects table for a significant period of time (say 10*MaxDelay) then A somehow arose in an erroneous way. (Scenarios which can lead to such are situation are complex and include such things as processes crashing during moments when various other processes have inconsistent information about what process owns what object.) In any event, P rectifies the situation by removing A from its world model. (If by some reason, A really should be in the world model, then it will eventually get into an Object State Summary and reappear.)

The above mechanism is included in ISTP as a last resort way to make sure that all processes eventually agree on what objects exist.

Summary interval. A critical parameter of the above is how often Object State Summary messages are sent. There is a trade-off between quickness of repairs and the bandwidth used.

If the summary interval is made very small (e.g., fractions of a second) then repairs will be made very quickly, but a significant amount of bandwidth will be used sending the summaries and perhaps worse, resources will be expended making repairs that are better off never being made, because they will soon be obsolete.

(The bandwidth used for Object State Summaries themselves could be reduced by not sending any summary at all when the server thinks that no objects have changed or been newly added. However, if this is done, then processes have to be prepared to reason from the absence of Object State Summary messages that information they are sending is not reaching S.)

If the summary interval is made very large (e.g., many seconds) then repairs will not be timely, but the bandwidth used for both summary messages and repairs is minimized.

The parameter MaxDelay is used to control the summary interval, because it makes sense for both intervals to be the same. From the perspective of remembering information about out of order messages, making MaxDelay larger has very little cost. On the other hand, handling out-of-order messages with lateness greater than the time explicit repairs are made has little if any value.

Only experimentation in a particular network and application environment can yield the best value for MaxDelay. However, we expect that, all things being equal, a value of one to a few seconds is best.

Reliability. Because Object State Summary messages are incremental, they ensure reliability only if they are themselves 100% reliably communicated. The code doing the TCP transport must take great care to ensure this. If any interruption in TCP communication occurs, this must be reported as a break in communication so that a complete restart and reinitialization can ensue. The 1-1 Connection subprotocol of ISTP is designed to provide this kind of ensured reliability.

22.9 Recovering Lost Information

UDP messages sent by a process P_j can either be lost on their way to S or on their way to another process P_k. Losses are repaired in different ways in these two cases.

Remotely owned objects. As noted in the last section, a process P_k determines that it has failed to receive one or more descriptions about an object A it does not own via Object State Summary messages. When this happens, P_k sends (via TCP) an Object State Summary message to S stating what P_k already knows. S replies (via TCP) with an Object State message containing appropriate differential object descriptions. (A single pair of messages suffices to update all lost information.)

The Object State Summary message sent by P_k is syntactically identical to the ones sent by S. It is also semantically identical in the sense that it is accurately summarizing what P_k knows about objects it does not own. However, it is used in a different way, because it is not used to update the objects table in S. Rather, S sends a message to P_k, to update the world model copy in P_k. Note also that the TopicID in the Object State Summary P_k sends is P_k's communication ID for the locale in question and that there are never any references to an object P_k owns.

Suppose that the most recent state information known to S for an object A corresponds to counter E. Suppose further that S receives an Object State Summary message from P_k specifying that P_k knows only about state C of A ($C < E \bmod 2^{16}$). In that case, S sends a description of state E of A to P_k that can be understood relative to C. P_k can use the state $C = 0$ to indicate that it knows nothing about A. This forces S to send a full description of S.

As messages travel between S and P_k, both S and P_k may be learning more about A from arriving UDP messages. That is all to the good. At any moment S and P_k respond based on the best information they have.

Locally owned objects. As also noted in the last section, a process P_j determines that one or more descriptions it has sent out about an object A it owns have not been received by S if too much time elapses without these descriptions being reflected in an Object State Summary message from S. Specifically, one of two cases obtains.

Case one: the highest counter value for A in the objects table is C while P_j knows that A is in state D ($D > C \bmod 2^{16}$). (Note this case includes the case that state D specifies that A has been removed.) In that case, P_j sends a description of A to S that can be understood given state C. This is sent in an Object State message over the TCP connection from P_j to S to guarantee that it will be received.

Case two: A is not in the objects table while P_j knows that A is in state D. (Note this case could also conceivably include a situation where A was removed already.) In that case, P_j sends a full description of A to S that can be understood given no prior information. Again, This is sent in an Object State message over the TCP connection from P_j to S to guarantee that it will be received.

The value of multi-state basis differential descriptions. It is important to realize that while the above is the method of explicit message repair in ISTP, it is not the only method of repair and in many situations not even the most important method of repair. In particular, information about rapidly changing objects is often rendered obsolete by descriptions that can be understood without reference to lost messages. This allows many repairs to effectively be made without taking any extra action other than creating differential messages that can be understood based on several prior states.

22.10 Reliability Control

In order to provide detailed application level control of the level of reliability vs speed in ISTP, every shared object is given the following two control bits.

Reliability and Synchronization. If the ForceReliable bit (which by default is off) is on in an object when a change is going to be communicated, then the change is communicated by TCP to the server, which in turn uses TCP to communicate the change to the other processes in the group. (This is the same kind of communication that is used when multicast has to be simulated. The ForceReliable bit must be shared, because the server needs to know when it is on.) This is slower than using multicast to communicate the information and requires greater bandwidth, but minimizes the time until every process in the group will know that the change has occurred. Several non-obvious aspects of this communication are important:

First, in general it is intended that this feature will be used sparingly. In some sense the whole purpose of ISTP is to make this kind of communication unnecessary.

Second, beacons (which are not discussed in this document) are always communicated in this style via Content-Based Communication servers. In many situations, beacons provide a more selective way to get highly reliable communication between processes.

Third, whenever TCP is used to communicate object changes (by a client or the server) all the objects that need to be communicated are communicated together at the same time by placing them in a single Object State message. This guarantees that all the changes will be received at the same time. As a result, the ForceReliable bit can be used for synchronization.

If several objects are changed together, and the ForceReliable bit is set on in each object, then all the changes will be communicated together and every other process will see the changes as a group, rather than piecemeal. Note that if UDP were being used, it would be difficult to guarantee this, because some changes could be received before others and a message with some of the changes in it could get lost.

Fourth, when TCP is forced by the ForceReliable bit, differential messages are used in the interest of minimizing bandwidth. However, in order to guarantee that the messages can always be decoded by the receiver, they have to be differential with respect to the last ForceReliable message if any, not just the last message. (The reason for this is that if the last message was not reliable, then some receiver may not have gotten the last message.)

As a result, if the ForceReliable bit is set on after having been off for a while, it is very likely that a full object message will have to be sent. This can make the cost of setting the ForceReliable bit quite high. (Objects have to have an associated field that specifies what the most recent reliably sent counter value was, or the system core must just use a full message whenever the prior message was not sent reliably.)

Fifth, when the ForceReliable bit is set on, it stays on rather than automatically being turned off. If you want it turned off again, you have to do that explicitly.

Minimizing bandwidth. If the InhibitReliable bit (which by default is off) is on in an object, then information about changes in the object are communicated by multicast, and the system minimizes the effort spent to ensure that the message will be received. In particular, when the server finds out about a change that has the InhibitReliable bit set, it does not include the new counter value in Object State Summary messages. (Note that the InhibitReliable bit must be shared, because the server needs to know when it is on.) This means that while processes will get multicast change messages, processes that happen to miss messages about changes with the InhibitReliable bit on will never know that they have missed anything and therefore will not expend resources trying to get this information. Several non-obvious aspects of this communication are important:

First, in general it is intended that this feature will be used sparingly. In some sense the key purpose of ISTP is to make reliable communication so cheap that there is no need to have unreliable communication. Nevertheless, setting the InhibitReliable bit is appropriate when sending something like very rapid position updates, where the information is so rapidly out of date that there is no point in making it reliable.

Second, there is complexity here in that if the server gets several changes only the last of which has InhibitReliable set, then it must include the counter for the next to last change in its next Object State Summary message. It can achieve this by not updating its objects table when it receives changes with InhibitReliable set.

Third, another complexity here is that when using InhibitReliable, one certainly also wants to use differential messages. However, these messages should be differential all the way back to the last reliable message so that they can always be decoded. Alternatively, processes have to ask for updates from the server when they are getting differential messages they cannot decode even when they are not getting Object State Summary messages that say that changes are occurring. (The latter is permitted by ISTP, but it would be easy to make a mistake and fail to support it.)

Fourth, even if InhibitReliable is set all the time for an object, some changes nevertheless have to be sent reliably. In particular, the initial creation of an object and its eventual removal are always communicated reliably. If these changes were not reliably sent, then processes could get completely mixed up.

Fifth, when the InhibitReliable bit is set on, it stays on rather than automatically being turned off. If you want it turned off again, you have to do that explicitly.

Sixth, merely not changing the counter value would have a similar effect to the InhibitReliable bit, but would neither allow differential messages to work right nor allow processes to treat descriptions with duplicate counter values as coming from duplicate messages.

Speed and reliability. If the ForceReliable and InhibitReliable bits are both set in an object, then changes are sent both by TCP and multicast UDP. This is costly of bandwidth, but guarantees minimum latency of communication and full reliability in minimum time.

23 Streaming Audio

An ISTP process initiates streaming audio output in a locale L whenever it owns an spAudioSource object in L through which an audio stream is being sent. An ISTP process initiates streaming audio input in a locale L whenever it owns an spObserving object in L with the Audio bit on, or an spObserving object in a neighbor of L with the Audio bit on and the IgnoreNeighbors bit off.

The development of good protocols for Streaming Audio is being vigorously pursued in many places. It is the desire of ISTP to take advantage of this work rather than compete with it. Therefore, ISTP specifies as little as possible about how Streaming Audio should be supported. It is intended that ISTP will be compatible with most, if not all, standard approaches to Streaming Audio. Only three requirements are imposed.

(1) An ISTP process P must be able to control when audio communication in a given locale should start and stop.
(2) P must be able to control what address will be used for audio communication. (Communication should be supported both by multicast and via simulated multicast through 1-1 Connections to a Locale-Based Communication server.)
(3) When receiving messages containing audio data, P must be able to identify which spAudioSource object the data is associated with.

Features of Streaming Audio that make it different from Locale-Based Communication include:

(1) Separate addresses are used for communication object data and audio data in a locale, so that a process can choose to attend to just one of these two kinds of data.
(2) No attempt is made to recover audio data that is lost. (Since ISTP focuses on real time interaction, the total time between when audio data is sent and used must be kept very small. As a result, it is very unlikely that lost data could be retrieved rapidly enough to be used.)
(3) When Streaming Audio reception is initiated, no attempt is made to initialize the receiver with old data. Rather, things are merely set up for the receipt of new data.

At the current time, Spline uses Lawrence Berkeley Lab's Visual Audio Tool (VAT) to support Streaming Audio. VAT in turn uses multicast RTP to communicate audio data between processes. This is an imperfect implementation of ISTP that causes Streaming Audio to be less integrated into the rest of communication than is desirable. As better tools become available, Spline will use them to support ISTP more completely.