quit

SYNOPSIS: quit

The quit command shuts down and deallocates the currently running agent.

The list command is used to select a group of resources from the SDS resource database. The list command with no arguments returns the entire SDS resource database as a list of resource IDs. There are two other forms of the command: select and join. The select version performs a logical AND operation (a resource meeting *all* of the selection criteria is returned); the join operation performs a logical OR (a resource meeting*any* of the selection criteria is returned). The criteria are in the form of key=value pairs (for example, “Type=Interpreter”) which must match attribute values in the resource. The command retuns a list of all resource IDs which match the selection criteria.

when

SYNOPSIS: when <predicate> do <action>

The when command allows you to define actions which will happen asynchronously at some point in the future whenever the given predicate is satisfied. the predicate will be installed in a list of predicates. Whenever the SDS server changes state, all of the predicates installed by “when” commands are evaluated. All predicates would return either 1 or 0. If a predicate returns 1 then its action is fired.

Typically both the predicate and the action will be the names of procedures which are evaluated when the server changes state. You can, however, specify predicates and actions “in-line” for convenience.

Actions should also return either a 1 or a 0. A return value of 0 indicates that the predicate-action pair should be removed from further consideration (this return value effectively dis-installs the predicate and action installed by “when.” A return value of 1 indicates that the predicate and action should remain installed.

pingme

SYNOPSIS: pingme

The pingme command generates a NULL callback (RPC procedure number zero) to the client which created the interpreter. This command is primarily useful for debugging the RPC callback mechanisms.

event

SYNOPSIS: event <type> [<key>=<value>]*

The event command generates an event back to the client which created the interpreter. You must specify the type of the event. Optionally, a list of key-value pairs may be provided. Each of these will create and set an attribute within the event before it is transmitted. Agents can use this command to easily generate custom event types.

shutdown

SYNOPSIS: shutdown

This command performs a clean shutdown of the Shared Data Service.

clear

SYNOPSIS: clear

The clear command empties the SDS resource database.

Appendix B: SDS Agent Scripting Language Specification

This section details the SDS agent scripting language. The basic TCL commands are not covered; only the new commands which directly deal with the SDS.

help

SYNOPSIS: help [<command>]

The help command is provided for interactive usage. It prints out a list of all commands with short summaries. If provided with a command name, it prints out detailed help information on that command.

create

SYNOPSIS: create <type> [<attr>=<value>]*

The create command allows for the creation of new resources. You must specify the type of the resource to be created. You may also specify any initial attributes for the resource in the form of key-value pairs (for example, “create MyType foo=goo”). The command returns the ID of the newly created resource.

destroy

SYNOPSIS: destroy <resourceid>

This command destroys the specified resource.

resource

SYNOPSIS: resource <resourceid>
resource <resourceid> get <attribute-name>
resource <resourceid> set <attribute-name>
resource <resourceid> exists <attribute-name>
resource <resourceid> exists

The resource command allows you to set and get attribute values, and check for the existence of resources and attributes.

In the first form, an entire dump of the resource’s attribute dictionary will be returned.

In the second form, an attribute name is provided and the command returns the attribute value (or NULL if it doesn’t exist).

In the third form, both an attribute name and value are provided, and the value is returned. The specified attribute is set to have the provided value. If the attribute doesn’t exist, it is created with the new value.

list

SYNOPSIS: list
list select [<attr>=<value>]*
list join [<attr>=<value>]*

```
SDS_CLEAR_SDS(void) = 8;
```

The MORE_INFORMATION proc is used by the server to ask the client to provide further information on a resource so that the server may “fill in” its database. Currently this call is unimplemented. It will probably be used in the future to maintain some sort of loose consistency between client-side copies of resources and the “master” copies maintained on the server.

```
void
SDS_MORE_INFORMATION(void) = 9;
```

The SOLICIT_INTEREST call is used by the client to inform the server about the types of events it is interested in receiving. It is currently unimplemented. This call is only valid on the server side.

```
void
SDS_SOLICIT_INTEREST(void) = 10;
```

This message defines an EVENT which is passed from the server to the client. It is only valid on the client side. The argument is the event which is sent.

```
void
SDS_EVENT(SDS_Wire_Event) = 11;
```

The NORMALIZE_FILEID proc is used by the client to “trade in” a client-generated file id for a “normalized” file id from the server. Normalized means that the ID is updated so that it may be used for between-hosts compares.

```
SDS_Wire_FileID
SDS_NORMALIZE_FILEID(SDS_Wire_FileID) = 12;
```

The INTERP proc passes a string to the specified interpreter for execution. The return value is the result of the execution, including status and result codes.

```
SDS_Wire_InterpRet
SDS_INTERP(SDS_Wire_InterpArgs) = 13;
```

The GOODBYE message informs the server that a given client is shutting down, and that all server-held resources associated with that client should be freed. It is only valid on the server side.

```
void
SDS_GOODBYE(void) = 14;
```

These are the actual SDS program and version numbers.

```
 } = 1;
} = 81966;
```

DESTROY_RESOURCES is used to destroy a list of resources by ID. The call returns a status list indicating the success or failure of each destroy operation. This call is only valid on the server side.

```
SDS_Wire_StatusList
SDS_DESTROY_RESOURCES(SDS_Wire_IDList) = 3;
```

The UPDATE_RESOURCES procedure is used to change the values of attributes on a resource. It can also add attributes to a resource. The argument provides for the updating of multiple resources at a time. The call returns a status list indicating the success or failure of each update operation. This call is only valid on the server side.

```
SDS_Wire_StatusList
SDS_UPDATE_RESOURCES(SDS_Wire_UpdateRecordList) = 4;
```

The SELECT proc is used to search for resources which match a set of provided search parameters. The argument provides a set of search parameters. The return value is a list of all resources which match the parameters. This call is only valid on the server side.

```
SDS_Wire_ResourceList
SDS_SELECT(SDS_Wire_SelectArgs) = 5;
```

The RESOLVE proc is used to “trade in” an ID for a full object. That is, given an ID the call returns a resource. This is useful for clients which need to match IDs to resources (name resolution), or for updating client-side caches. The argument is a list of IDs; a list of resources is returned.

This operation is bidirectional. The client can resolve an ID for the server at the server’s request. It’s not entirely clear how useful this may be.

```
SDS_Wire_ResolveRet
SDS_RESOLVE(SDS_Wire_IDList) = 6;
```

SHUTDOWN gracefully terminates the Shared Data Service. This call is authenticated on the server side. Eventually it will be restricted to clients with the proper authorization. This call is only valid on the server side. It returns a status indicating whether the server was successfully shutdown or not.

```
SDS_Wire_Status
SDS_SHUTDOWN_SDS(void) = 7;
```

The CLEAR proc clears and resets the SDS database, leaving it in an “empty” state. This is primarily useful only for debugging. This call is authenticated on the server side; eventually it will be restricted to clients with the proper authorization. This call is only valid on the server side. It returns a status indicating whether or not the server was successfully cleared.

```
SDS_Wire_Status
```

This structure is the return value from SDS_RESOLVE. It gives a status indicating how the operation completed for each input ID, and a list of resources to satisfy the resolve operation.

```
struct SDS_Wire_ResolveRet {
 SDS_Wire_StatusList status;
 SDS_Wire_ResourceListresources;
};
```

This datatype is the argument to the SDS_HELLO operation. It is used to establish a client's identity, and to provide information which can be used by the server to contact the client (via RPC callbacks). Host is the host from which the client is calling, prognum is a transient RPC program number which the client is using for callbacks. TransportName is currently unused.

```
struct SDS_Wire_HelloArgs {
 varstring           host;
 varstring           transportName;
 long                prognum;
};
```

Protocol Message Definition

The RPC program number is SDSPROG, and the RPC version number is SDSVERS.

```
program SDSPROG {
 version SDSVERS {
```

The NULL procedure "pings" the server. Clients implementing RPC callbacks also respond to this request.

```
void
SDS_NULL(void) = 0;
```

The HELLO proc is used to establish communication with the SDS server. It provides the information necessary to generate server-to-client callbacks. This proc is only valid on the server side.

```
SDS_Wire_Status
SDS_HELLO(SDS_Wire_HelloArgs) = 1;
```

CREATE_RESOURCES is used to create any number of resources of any type in the SDS server. The return type includes a list of statuses indicating the success or failure of each creation, along with the IDs of the newly-created resources. This call is only valid on the server side.

```
SDS_Wire_CreateResourceRet
SDS_CREATE_RESOURCES(SDS_Wire_ResourceList) = 2;
```

Protocol Argument Types

SDS_Wire_SelectArgs is the argument type for the SDS_SELECT protocol request. It defines a search mode (SELECT or JOIN), along with a set of keys and values to search for. The flags member is currently unused.

```
struct SDS_Wire_SelectArgs {
 int             searchMode;
 SDS_Wire_KeyListkeys;
 SDS_Wire_AttributeListattribs;
 int             flags<>;
};
```

SDS_Wire_UpdateRecord is the base argument type for the SDS_UPDATE_RESOURCES protocol request. It takes the ID of the resource to update, and lists of key and values to be updated.

```
struct SDS_Wire_UpdateRecord {
 SDS_Wire_ID     id;
 SDS_Wire_KeyListkeys;
 SDS_Wire_AttributeListattribs;
};

typedef struct SDS_Wire_UpdateRecord SDS_Wire_UpdateRecordList<>;
```

SDS_Wire_InterpArgs defines the datatype which we pass to SDS_INTERP. Basically it is the ID of the interpreter to which the data is destined, and a string consisting of command data to be interpreted.

```
struct SDS_Wire_InterpArgs {
 SDS_Wire_ID     id;
 varstring       str;
};
```

This structure is the return type from SDS_INTERP. The status indicates a global success or failure value, the code is the return code from the INTERP operation, and the string is the output string of the INTERP operation.

```
struct SDS_Wire_InterpRet {
 SDS_Wire_Status status;
 int             code;
 varstring       str;
};
```

This structure is the return value from SDS_CREATE_RESOURCES. It gives a list of statuses indicating how the create operation fared on each input resource, and a list of IDs of the newly-created resources.

```
struct SDS_Wire_CreateResourceRet {
 SDS_Wire_StatusList status;
 SDS_Wire_IDList     ids;
};
```

Wire types are never visible to the application developer.

The protocol definition defined here includes both messages which are designed to be sent from the client to the server, and messages sent from the server to the client (such as callback and event messages). The protocol definition indicates the “direction” for which each request is valid.

Fundamental Data Types

Strings and stringlists are used as subtypes in a number of protocol data types. Both are variable-length. SDS_Wire_FileID is used to uniquely identify a file, network-wide. SDS_Wire_IDs and SDS_Wire_Keys are simply strings, while SDS_Wire_Statuses are integers. SDS_Wire_Attributes currently consist of a string value and the immutable flag.

```
typedef string varstring<>;
typedef varstring stringlist<>;

struct SDS_Wire_FileID {
 stringhostname<>;
 stringdevice<>;
 longinode;
};

typedef varstring SDS_Wire_ID;
typedef SDS_Wire_ID SDS_Wire_IDList<>;

typedef varstring SDS_Wire_Key;
typedef SDS_Wire_KeySDS_Wire_KeyList<>;

typedef int  SDS_Wire_Status;
typedef SDS_Wire_Status SDS_Wire_StatusList<>;

struct SDS_Wire_Attribute {
 varstringvalue;
 int     immutable;
};
typedef SDS_Wire_Attribute SDS_Wire_AttributeList<>;
```

SDS_Wire_DictClass is the wire datatype used for both resources and events: it defines a dictionary of key and attributes. Thus, resources and events are essentially the same. We typedef SDS_Wire_Resource and SDS_Wire_Event to this dictionary type, and create variable length lists of both types.

```
struct SDS_Wire_DictClass {
 SDS_Wire_KeyListkeys;
 SDS_Wire_AttributeListattribs;
};

typedef SDS_Wire_DictClass SDS_Wire_DictClassList<>;

typedef SDS_Wire_DictClass SDS_Wire_Resource;
typedef SDS_Wire_DictClass SDS_Wire_Event;

typedef SDS_Wire_Resource SDS_Wire_ResourceList<>;
typedef SDS_Wire_Event SDS_Wire_EventList<>;
```

Attribute	Description
Object ID	Identifier of an Object resource.
Data ID	Identifier of a Data resource.

TABLE 7. Standard attributes of Activity resources

Extensions to the Scripting Interface

At some point, Intermezzo will provide a set of extensions to the agent scripting interface which provide higher-level mechanisms for direct operations on Intermezzo constructs. These extensions will deal directly with Users, Tasks, Objects, Data, and Activities, instead of the simple resource-level facilities provided by SDS.

Currently, however, no work has been done on extensions to the scripting interface.

Caveats

There are several weaknesses in the current Intermezzo implementation. This section describes the most obvious of these weaknesses.

- Singly-threaded

  The current implementation of the Shared Data Service is singly-threaded. This limits its performance over what could be obtained through a multi-threaded implementation. Further, the server may appear to "hang" if a client is executing interpreted code in an agent which goes awry.
- Non-distributed

  The SDS is currently implemented as a centralized, non-distributed service. This limits reliability and scalability of the system. The centralized implementation does make the system much easier to implement however (for example, there is no need to distribute agent interpreter state across multiple machine).
- Unimplemented features

  There are still a number of unimplemented features in the system. Protection and persistence across SDS runs are two of the most important. These will be addressed in a later version of the code.

Appendix A: SDS Protocol

This section describes the RPC protocol used for communication with the Shared Data Service in depth. The protocol description is given as excerpts from the RPCGEN RPC Language specification file.

Note that the data types used here are very similar to the types visible to the application programmer (such as SDS_Resource, SDS_Attribute). These types, however, are wire types. Wire types are suitable for transmission over the network. The individual SDS object classes can construct themselves from wire types (and can similarly "unparse" themselves as wire types for transmission over the network).

Table 6 presents the core attributes which are present on Object instances.

Attribute	Description
Subtype	The Subtype of this Object (File, Database, etc.)
Subtype ID	A string which uniquely identifies this object within the space of its Subtype.

TABLE 6. Standard attributes of Object resources

Data

Data resources are used as generic “placeholders” for shared data by Intermezzo. An application can create a Data instance and publish its ID; it may then be accessed by other applications as a repository for shared data. In essence, the Data resource provides a generic data exchange mechanism (much like ToolTalk or RPC) which is built using the same mechanisms and interfaces as the rest of Intermezzo.

Every Activity instance has associated with it a Data instance which can be used for data interchange with that Activity (see below). Currently, however, no Intermezzo clients actually make use of the Data resource.

Activities

The Activity resource is perhaps the central resource type in Intermezzo. It is a container resource which groups together information about a single user, working with a single application, on a single object. The Activity resource contains the IDs of User, Task, Object, and Data resources. It may be thought of as a single row in a database containing all user activity across the network.

Typically, when an Intermezzo-aware application starts up, it creates a single instance each of a Task, Object, and Data resource. It may also optionally create a User instance, although it has the option of finding an already-existing User instance which represents the human user of the application. Once these resources have been created and published, Intermezzo will create an Activity instance containing the IDs of these other resources. This Activity instance says, in effect, “this user is working with this application on this data object.”

The Intermezzo client-side API supports operations in terms of Activities. For instance, an application may issue a query asking for all Activities in which the Object matches some value (say, a file that the application is preparing to open). Another application may issue a request for all Activities in which a certain user is involved.

While Activities are the semantic backbone of the Intermezzo database, they are internally very simple. Their only attributes are the IDs of their member objects. See Table 7.

Attribute	Description
User ID	Identifier of a User resource.
Task ID	Identifier of a Task resource.

TABLE 7. Standard attributes of Activity resources

Attribute	Description
Environment	A concatenated string representing the environment under which the process is being run (retrieved from envp).
Resource Utilization	Resource utilization statistics of the application (currently unused).

TABLE 5. Standard attributes of Task resources

Objects

Objects represent the “artifacts” of collaboration. That is, they are the particular “thing” on which an application (as represented by a Task) operates. The term *object* here is used in the sense of *direct object* (what the application operates on), as opposed to its meaning in the object-oriented programming field.

Objects store information about files, database regions, or other sharable artifacts of collaboration. One of the central ideas in Intermezzo is that session management can be made implicit by being able to uniquely determine when two users are operating on the same object. Thus, Object instances need to carry enough information to uniquely identify themselves, even when used in a network-wide context.

Of course, different applications operate on different objects. A text editor may operate on a file; another application may operate on a segment of a database. In a video conferencing program, the other user may be considered the object of the application. Thus, Intermezzo needs to be as flexible as possible in allowing the specification of Object instances. There are a potentially limitless array of different kinds of objects (files, databases, etc.), so Intermezzo uses an object Subtype attribute to identify more specifically the type of the Object. A Subtype ID attribute provides an ID string which should uniquely identify the object within the space of its Subtype, across the entire network.

It is the responsibility of the client-side interface to be able to generate a unique Subtype ID for a given Subtype. Obviously, since there are a potentially large number of Subtypes, the client-side API will never be able to handle all Subtypes. Convention will dictate which Subtypes are supported; application developers should agree on a limited set of Subtypes.

Currently Intermezzo only supports one Subtype: the File. Code in the Intermezzo API can generate a network-wide unique Subtype ID for files based on the host on which the file resides, the filesystem on which the file resides, and the inode of the file. This ID is independent of the name used to refer to the file and can be compared with IDs generated on other hosts.

Note that we have side-stepped the issue of granularity here. If a user is editing a file with a text editor, the Object for that text editor will represent the entire file. It may be useful in the future to have more fine-grained information stored in Objects. For example, which paragraph in the file is the user currently in. It should be possible to integrate attributes into the Object resource which provide more fine-grained specification of regions of interest.

Attribute	Description
Work Phone	User's work phone number (taken from WORKPHONE environment variable).
Home Phone	User's home phone number (taken from HOMEPHONE environment variable).
Email Address	User's email address (taken from 'aliases' NIS database).
Plan	User's "plan" information (taken from $HOME/.plan).
Project	User's "project" information (taken from $HOME/.project).
Face	User's facial image, in XFACE format (taken from $HOME/.face).

TABLE 4. Standard attributes of User resources

Of course, since resources are extensible, users and applications can insert new attributes into a User object as needed or desired. For example, a user may insert a new attribute with a voice clip of themselves. Applications may associate user preference data with their User object instances.

Tasks

Tasks are representations of applications running on the system. Each Task object maintains information about the context and status of a running application. The Intermezzo client-side library provides convenience routines for generating "minimal" Task object instances, given basic environmental information (such as argc, argv, and envp). The library is designed to make it easy for an application to publish a Task instance which represents that application itself (that is, the Task generation routine is designed to be run from within the context of the application which the Task is representing).

The library generates certain attributes automatically. Table 5 lists those attributes. Of course, Intermezzo clients are free to add attributes as needed.

Attribute	Description
Application Name	The name used to start the application (retrieved from argv[0]).
Start Time	The time of application start-up (actually the time of creation of the Task instance).
Idle Time	Time since the last user input (this is currently always zero).
Process Host	The host on which the application is being run.
Display	The X display to which the application is sending output.
Controlling TTY	The controlling TTY of the process.
PID	The process ID of the application.
PGID	The process group ID of the application.
Real UID	The real user ID under which the process is being run.
Effective UID	The effective user ID under which the process is being run.

TABLE 5. Standard attributes of Task resources

A few more advanced features are present in the scripting language however. One is the deferred or asynchronous procedure call. This feature allows some procedure call to be run whenever some predicate condition is satisfied. The agent itself is not even notified directly when the procedure is called (unless, of course, the procedure takes some explicit action to notify the agent).

Appendix B gives a detailed overview of the agent scripting language.

Intermezzo-Specific Resources and Facilities

Introduction

This section describes facilities which are specific to Intermezzo itself (rather than, say, a part of the Shared Data Service). Remember that Intermezzo is built on top of the SDS: it is an SDS client and all SDS facilities are available to it. This section describes how the features of Intermezzo interact with the SDS.

Intermezzo uses the storage management facilities of the SDS to create and *publish* objects which contain information useful in the process of collaboration. The publish metaphor is used because any interested party can *subscribe* in order to be informed of any changes in the state of published objects.

The Intermezzo client-side API uses this publish/subscribe metaphor.

Intermezzo Resources

Intermezzo uses the SDS resource typing facility and provides five distinct types of resources: Users, Tasks, Objects, Data, and Activities. Each of these resource types store a certain category of information. Each is described below. Note that in the descriptions of these resources, the SDS Core and Semantic attributes are not listed. The Core attributes are always present, and the Semantic attributes are optional (as they are with all SDS resources).

Users

User objects store information about the human users of the system. The Intermezzo client-side interface provides routines for generating User objects with some simple attributes set. At a minimum, Intermezzo ensures that certain attributes will be present on all instances of Users. Table 4 lists those attributes.

Attribute	Description
Login Name	User’s login name (taken from password database).
Real Name	User’s real name (taken from password database).
Home Directory	User’s home directory (taken from password database).
Shell	User’s preferred shell (taken from password database).
Office	User’s office location (taken from OFFICE environment variable).

TABLE 4. Standard attributes of User resources

- *Flexible Event Solicitation*

  Intermezzo clients use an event-driven programming model. When something "interesting" happens at the server, clients are notified via an event. The problem comes in determining what defines "interesting." Typically, APIs provide a procedure call which is used to inform the server about the types of events the client is interested in. This procedure is called event solicitation.

  The problem with using a procedure call to do event solicitation is that it may be difficult to fully specify what clients are interested in via parameters to a procedure. There may not be enough flexibility in procedure calls, especially in very robust, data-rich environments.

  By embedding interpreters in the shared data service we gain a great deal of flexibility. An agent running in an interpreter can perform an arbitrarily complex evaluation of the "interestingness" of an occurrence before deciding to send an event to a client.

- *Efficiency*

  Embedding computation in the server itself provides a great deal of efficiency in terms of network utilization. A good analog is the lower network usage of NeWS as compared to the X Window System. A portion of an application runs within the server itself; communication between this portion of the application and the server itself is quick and utilizes no network resources.

- *Tools for Application Development*

  A programmable data storage engine provides a powerful model for application development. Applications are built as two cooperating parts: one part running locally on the client host (probably written in C or C++), and another part running within the server itself. The portion of the application running in the server is written in a very high-level interpreted language which allows for easy prototyping.

  Application writers can split their applications along logical lines. A great deal of the functionality of the application can be developed very easily using the scripting interface.

- *Powerful Model for Interaction*

  Perhaps most importantly, agents provide a very powerful model of interaction. Users and applications can define autonomous agents which act on their behalf, informing them of situations which may interest them, aiding them in their computation, and generally providing data sharing services to them.

  A user may typically have a number of agents running at a given time; some may be loaded by applications solely to perform some application-specific function, and thus "hidden" from the user (the user is never contacted directly by the agent). Other agents may be scripted by users themselves to handle some chore for them (such as popping up a message on their screen when one of their coworkers arrives in the morning).

Scripting Language

The agent scripting language provides direct access to SDS functions. There is essentially a one-to-one mapping between scripting language commands and the SDS protocol. The basic TCL language provides features such as variables, arrays, lists, control flow, and exception handling.

The server itself is approximately 5000 lines of C++ application code (this does not include the roughly 1100 lines of C code which are automatically generated by RPC-GEN to implement the protocol itself, nor does it include code which implements the data types stored by the SDS itself).

The data type code, along with storage classes for those data types, is approximately 5200 lines of C++. This code is used by both the server and the client-side interface to implement resources, attributes, IDs, and other classes which represent the actual database stored by the SDS.

The client-side interface is approximately 3100 lines of C++.

Summary

It is expected that clients will use the facilities provided by the Shared Data Service to build up higher-level object semantics on top of the basic data storage facilities provided by the SDS. As an SDS client, Intermezzo imposes creates and manipulates collections of resources. These resources have Intermezzo-defined types and attributes which provide utility in the domain of collaboration support.

SDS Agent Scripting Interface

Overview

As mentioned previously, the Shared Data Service allows clients to create interpreters internal to the SDS server itself. These interpreters can run scripts (called *agents*) which have full access to the facilities of the Shared Data Service.

Interpreters are accessed as normal resources. To create an interpreter, a client tells the SDS to instantiate a new resource with the Type attribute set to “Interpreter.” The SDS detects the attempt to create an Interpreter resource, instantiates a new interpreter internally, and adds some attributes to the new resource to allow clients to access interpreter internals. Code run in an interpreter runs with the permissions and privileges of the client which created it.

The INTERP protocol request is used to pass a string for execution to a named interpreter. The results of an INTERP operation return two values: a status code and a result string.

The agent scripting language used by the embedded interpreters is based on Tool Command Language (TCL), with a number of extensions to allow access to SDS internals. This section explains why agents are useful, and provides some detail on the scripting language itself. See Appendix B for full details on the scripting language.

Why Embedded Interpreters?

Why are embedded interpreters useful anyway? There are a number of reasons:

sible. It is very easy to create a new type of event for an application-specific circumstance.

The simplest way for clients to receive events is to *solicit* for specific event types from the SDS. Clients can tell the SDS that they wish to be informed whenever some particular state change occurs in the server (for example, when a new resource of a given type is created). Each protocol request to the server triggers a state change, and at each state change client event interest lists are evaluated.

If clients wish to receive events based on more complex circumstances (for example, some series or pattern of state changes which cannot be expressed simply, or some occurrence based on some complex set of resource attribute values), then clients can create an agent which looks for this set of complex circumstances to be satisfied (via an asynchronous procedure, see above) and then generates an event back to the client when this happens.

The client-side programming interface allows applications to install event handlers based on the particular type of the event.

Communications Architecture

The Shared Data Service is currently implemented using Transport Independent Remote Procedure Calls (TI-RPC), a part of the ONC+ networking package. The service runs as a single, centralized process, rather than as a true distributed system (see “Rendezvous,” below).

The system is implemented so that both clients and the server can use the NETPATH environment variable to set the transports on which connections can be made. SDS currently can use only connection-oriented transports, such as TCP/IP. Thus, “circuit_n” semantics are used for transport selection.

Rendezvous

Rendezvous is the problem of how clients initially connect to a facility available somewhere on the network. In true distributed systems, client typically connect to a nameserver program which runs on the local host. This nameserver then resolves the desired network resource for the client.

The current SDS implementation is not distributed. There is only one centralized instance of the SDS per collaborative “universe.” We currently have no sophisticated rendezvous facilities for allowing clients to find the SDS. It is expected that the location of the SDS server will be a well-known host; the name of this host is provided to the clients at start-up time.

If future implementations of the SDS are truly distributed it is expected that more sophisticated rendezvous facilities will be required.

Implementation

The Shared Data Service is implemented in C++ using TI-RPC. The Rogue Wave Tools.h++ classes are used to provide storage classes and several miscellaneous other classes (such as strings and dates). (Tools.h++ is a commercially available class library for C++ developers.)

SDS Agents

The SDS provides one unique feature which is not found in many object databases. The Shared Data Service supports multiple embedded interpreters which can access and modify data stored by the SDS. When designing the SDS it was clear that there was a need to be able to generate events back to the client based on arbitrary occurrences within the SDS. Rather than attempting to build an API which could capture the numerous conditions under which events should be generated, it was decided that embedded interpreters could be an effective way of providing flexibility and power to SDS clients.

Clients can instantiate interpreters within the SDS and download code into these interpreters. Each interpreter runs with the permissions of the client which created it. Code downloaded into the SDS can modify resource data, create resources, pass data to other interpreters, or generate events back to SDS clients (see the section on “Events” below).

These interpreters are effectively *agents* which run within the SDS, performing services on behalf of the client which created them. The agents can run within the SDS looking for some set of arbitrarily complex circumstances to occur, and then inform the client when they take place. Agents allow us to specify very complex behavior to the SDS without having to encode this information into a set of parameters to a procedure call.

Agents have access to the full power of the SDS and thus can be very flexible. A typical application which makes use of the SDS may be divided into two portions of code: one part which runs on the local host (probably implemented in C or C++), and another portion which is written in the SDS Agent Scripting Language and runs in the SDS itself. These two portions of code cooperate together to provide the functionality of the application.

One of the more powerful features of agents is the availability of “asynchronous procedures.” An agent can provide a procedure which will only be run when some predicate set of conditions is satisfied. Whenever there is a state change within the server, the SDS evaluates the predicates for all asynchronous procedures which are present. If any of these predicates evaluate to True, the procedure associated with that predicate is fired without any intervention from the agent script itself. Agent programming is thus simplified from the typical “event loop” model (sit in a loop, waiting for something interesting to happen), into a model in which the agent program simple describes the occurrence it is interested in, and the server itself takes the specified action when needed.

Currently the interpreters within the SDS understand Tool Command Language, or TCL. TCL is a simple scripting language designed to be embedded in applications. The TCL language used by the SDS has been enhanced with several new primitives to support operations on resources and attributes. One weakness of the current implementation is that the entire Shared Data Service runs as a single thread of control. Thus, it is possible to “hang” the SDS by downloading buggy code into an interpreter.

Events

Clients are notified of changes in the SDS via *events*. Events are messages which are sent from the Shared Data Service to clients. Events can be generated as a result of some native SDS operation, or by code running in any one of the SDS interpreters.

Events themselves are quite similar structurally to resources: they are basically just dictionaries of key-attribute pairs. Every resource must have an ID and a Type. Other event fields are available for use by the SDS itself or SDS agents. Thus, SDS events are exten-

care, since indiscriminate use of it may cause the SDS data store to grow very large). (See the section, “Persistence,” below.)

The Protection attribute defines the visibility and writability of the attribute’s value. Strings can be loaded into this attribute to allow or disallow other clients from reading or writing the resource’s data. (See the section, “Protection, Privacy, and Authentication,” below.)

The Type field can, in general, be used freely by clients to provide data typing to the objects stored in the SDS. The SDS itself defines one special value for the Type attribute however. If the value of Type is “Interpreter” then the SDS will recognize this resource as a “place-holder” for an internal embedded interpreter (agent). Interpreter resources have a few other unique attributes which are used and created directly by the SDS. These include Interpreter Input, Interpreter Output, Interpreter Name, and Interpreter Value. The section “SDS Agents,” below, provides more detail on how Interpreter resources work. Representing interpreters as resources allows us to extend the power of the SDS without creating new abstractions; basically everything in the SDS is a resource, even interpreters.

Persistence

The Shared Data Service looks for the key “Persistent” on the resources it manages. If such a resource exists, and its value is “True,” then the SDS will ensure that the resource survives across client shutdowns. By default, when a client disconnects from the SDS, any resources it has created are destroyed along with it. The Persistent attribute allows resources to be long-lived.

Truly long-lived persistence (life of resources across runs of the SDS) is currently not implemented. All data is saved in memory only. If the SDS crashes all resource information will be lost.

Protection, Privacy, and Authentication

[Currently protection attributes are not implemented by the Shared Data Service.]

The Shared Data Service allows clients to set an attribute named “Protection” on each resource. This attribute describes the permissions the resource has with respect to operations from other clients (for instance, is this resource visible to other clients, and if so, which clients). The SDS will use this attribute to limit the access to resources from certain clients. Eventually, the protection attribute will be used to implement full access control lists, so that individual clients or users may be granted specific permissions on resources.

Since collaborative applications by their very nature involve the sharing of data among multiple users, there is a need for mechanisms to provide privacy. The SDS protection attribute can be used to ensure privacy. The SDS uses RPC DES (data encryption standard) authentication to “prove” that users are who they say they are. Each client application to the SDS runs with the permission of a certain user; the identify of that user is authenticated and recorded by the SDS and used along with protection attributes to determine if a client application possesses the permission to perform some particular operation.

unique client identifier and a resource identifier which is unique within the client which created it.

The Type attribute is provided in the expectation that SDS clients will use it to build up higher-level object semantics on top of resources. The SDS itself doesn't make use of this field, with one exception (see "SDS Agents," below).

The Creator attribute identifies the client which created this resource. The SDS uses this attribute to know when to "clean up" after a client when it has disconnected.

In general, developers writing to the SDS API never have to explicitly set the Id or Creator attributes; these will be set by the SDS itself. Developers are free to use the type field at their discretion, if storage of typed resources "makes sense" in their particular applications. If a type is not provided for a resource at creation time, the type defaults to "Generic."

Semantic Attributes

In addition to the core attributes, there are other attributes which the SDS will look for on resources. These attributes are not core attributes, because the SDS does not require them to be present (thus they do not have the `immutable` flag set), but they do provide extra semantic information to the SDS. These attributes are called *semantic attributes* because they have meaning to the SDS and are used directly by the SDS to perform some function, even though they are not required to be present. Table 3 provides a list of the currently-used semantic attributes.

Attribute Name	Used Values	Description
Persistent	True	The SDS will retain the resource in persistent store.
	False	The resource is transient; the SDS will not retain it in persistent store.
Protection	<string>	Defines the protection attributes of the resource: who may see it, who may alter it.
Type	Interpreter	Indicates that this resource represents an internal SDS interpreter.
Interpreter Name	<string>	The name of the interpreter (used in conjunction with Type = Interpreter).
Interpreter Input	<string>	The input string to the interpreter (used in conjunction with Type = Interpreter).
Interpreter Output	<string>	The output string from the interpreter's last result (used in conjunction with Type = Interpreter).
Interpreter Value	<integer string>	The status of the last interpreter operation (used in conjunction with Type = Interpreter).

TABLE 3. SDS Resource Attributes

The Persistent attribute indicates whether a particular resource will outlive the client which created it. By default, when a client disconnects from the SDS, all resources created by that client are freed. If the Persistent attribute has a value of True, then the resource will be retained after client termination (clients should use this option with

The Shared Data Service

Introduction

Intermezzo is built on top of a communications substrate called the Shared Data Service, or SDS. The SDS is essentially a very simple object database. The SDS itself does not provide any direct support for collaboration; it only provides a model for data storage and access. Intermezzo uses the facilities provided by the SDS to implement services specifically designed for collaborative applications.

The SDS provides data storage, persistence, name resolution of data objects, protection and privacy of data objects, and authentication. The SDS also provides an embedded interpreter which can be used to create scripts or *agents* which reside in the SDS and perform services on behalf of clients.

Resources and Attributes

The Shared Data service provides storage for objects called *resources*. A resource is simply an object which is stored by the SDS and can be made available to any clients of the SDS. Resources have associated with them a collection of *attributes*. Attributes are named pieces of data. For example, the attribute named “ID” may contain the ID of the resource. The name of an attribute is called a *key.*

Currently, both keys and the data stored in attributes (the *values* of attributes) can only be strings.

With this data model, resources can be thought of as dictionaries which map from Key strings to Attribute objects. The Attribute objects themselves contain a Value string and, perhaps, some other information. In the current implementation, attributes maintain a flag called `immutable` which indicates whether or not the attribute value may be modified after it is set. Essentially, however, resources are simply lists of key-value pairs.

Core Attributes

The Shared Data service is basically a free-form data storage engine. It does, however, impose a few restrictions on stored data. Mainly, there are a small number of *core* attributes which must be present in every resource, and cannot be altered once the resource has been created. Table 2 provides a list of these core attributes.

Attribute Name	Description
Id	The universal identifier of the resource.
Type	The type of the resource.
Creator	Identifies the client which created this resource.

TABLE 2. Core Resource Attributes

These attributes are set upon the creation of the resource and are flagged as immutable. Thus, they can never be changed or removed after creation time.

The Id attribute is used to uniquely identify a resource. Resource Id values can be used within the SDS server and across clients. The Ids themselves are a tuple containing a

Expression of Policy and Procedure

Intermezzo provides facilities to both applications and users for the expression of policy and procedure related to collaboration and sharing. The terms “policy” and “procedure” can entail a great deal: application default information, user preferences, across-application settings, and so forth.

The goal of the Intermezzo policy and procedure facilities is to provide mechanisms through which both applications and users can express complex desires regarding system behavior. In addition to the underlying mechanisms for supporting policy and procedure, Intermezzo also provides a language for expressing policy and procedure desires to the system easily.

The Intermezzo mechanisms for expression of policy and procedure are implemented as embedded interpreters within the object storage manager. Applications can create these interpreters at will. Further, applications (and users) can directly download code into the object storage manager. This code will run in the server and can act as an *agent*, performing services on behalf of the user or application.

Agents have full access to the database of shared Intermezzo collaboration and activity information. They can create, destroy, and modify objects (given that the clients which created them have the proper permissions of course). Further, they can generate events to any Intermezzo application. They can even invoke arbitrary programs for execution.

Application developers can use these facilities to program the behavior of their systems; this behavior can be easily changed if needed. For example, an application may download code into the server which constantly monitors for some set of conditions to be met (for example, a shared text editor may create an agent which watches for others who may be editing the same text file). When these conditions are met, the agent can take some action (for example, notify both applications involved in editing, or start a video-conferencing session between the two users).

Users can use these facilities to express their policy desires to single applications or to all applications. For example, if a user wishes to be notified when a person arrives at their desk, they may download an agent into the server which “watches” for that person, and notifies the user when he or she logs in in the morning.

By establishing conventions for operation between applications, it is also possible to establish “global” (across application) policy. For example, a user may not wish to be bothered during an important meeting, and thus may download code into the server which says, in effect, “if anyone tries to access me, inform them that I am currently unavailable.” It should eventually be possible to express very complex policy on the order of, “don’t bother me when I’m in the lab, unless it’s my boss, or unless it’s anyone attempting to go over the quarter financial data with me.”

Currently, almost no applications have sophisticated policy controls such as this; usually policy choices are on the order of a binary switch (“It’s OK to bother me” / “It’s NOT OK to bother me.”) Intermezzo can provide much more flexible policy controls to applications and users. Further, because all applications use Intermezzo for their policy control, it is possible to establish *global* policy for *all* applications using a single set of facilities (rather than having to use whatever application-specific policy controls exist for the set of applications the user uses).

user representation provide a single point for applications to retrieve user attributes; rather than searching through password files, “dot” files, and so forth for user attributes, Intermezzo allows the storage of any user-related information in a manner which is accessible by all interested applications.

One benefit of this is that users can update their representations once in Intermezzo and all client applications will see the update. This is in contrast to the model where each application stores its own user model, and users must interact with each application individually in order to update their representations globally.

Intermezzo stores user representations in the `User` object described earlier. `User` objects can store virtually any user attribute which may be expressed as a string or data. A simple programmatic interface allows applications (with the proper permissions) to retrieve or even set attributes in the `User` object. Furthermore, it is possible to extend the `User` object by adding new attributes. These attributes may contain application specific data. Thus, rather than maintaining per-user information internally, collaborative applications may find it useful to “publish” their information globally via Intermezzo. By making such information widely available it is hoped that conventions regarding interchange of user information will arise.

As a very simple application of user representation information, consider a new version of the “finger” program which could be written based on Intermezzo. This new application would have to make only one call to Intermezzo to retrieve all known information about the user. Contrast this to the current finger implementation which accesses the password database and various “dot” files to provide user information, and yet is not extensible to new information fields (such as a picture of the person in question, or their current calendar).

Data Sharing

A fourth goal of Intermezzo is to provide facilities for easily sharing data. As mentioned earlier, Intermezzo is not intended to be a full-fledged application information sharing toolkit; its forte is sharing of “between application” or environmental information. Still, the mechanisms provided by Intermezzo can be used for the sharing of arbitrary data between applications.

Intermezzo is built on top of a simple object storage manager, called the Shared Data Service (this storage manager is described in greater detail later in this paper). While Intermezzo typically uses the SDS for sharing of information concerning user activity and session management, applications also have direct access to the facilities of the SDS and can use the service as a general-purpose data sharing system.

Thus, applications with relatively simple data sharing requirements can use Intermezzo to facilitate data sharing among a number of processes (perhaps running on different hosts across the network). This approach may be easier than using other data sharing abstractions such as a separate application data sharing toolkit, if the application does not need the full power of such a toolkit. The facilities provided by Intermezzo also mean that developers do not have to build their own sharing facilities (perhaps on top of RPC, NIS+, or ToolTalk) for their simple data sharing needs.

mation will be reported in the form of an event containing relevant information about the confluence. “Collaboration aware” applications may then decide to give the users the opportunity to collaborate (say, to enter a shared editing mode on the file), or may simply notify the users that they are editing the same file (perhaps for the awareness value of the information).

“Collaboration naive” applications of course can do nothing with this information since they have been written as single-user applications. But it is possible to configure Intermezzo to notify some third agent when the potential for collaboration is detected. For example, a shared window system may be notified so that it can multiplex the input and output of textedit, in effect making it shared. Or a simple collaboration monitor program may just display the face of the other person editing the file.

ID	User	Task	Object
1	Keith Edwards	mailtool	/var/spool/mail/keith
2	Keith Edwards	textedit	~/foobar
3	Tom Rodriguez	nethack	<NULL>
4	Tom Rodriguez	textedit	~/.login
5	Tom Rodriguez	textedit	/hm10/keith/foobar

TABLE 1. A Subset of an Example Activity Database

Intermezzo includes code to determine that simple types of Objects are actually the same. In the example above, the two Objects referred to files (and the fact that they refer to files, along with the filenames themselves are published by the activity monitoring portion of Intermezzo). Intermezzo can determine whether two files Objects are the same, regardless of the machine the reference is made from or the filenames used to refer to the files.

Currently only Objects with the file “subtype” are recognized by Intermezzo. It should be possible to support additional Object subtypes in the future though (for example, to determine whether users working in the same part of a database are actually overlapping their accesses). Perhaps “equivalence detectors” for these new subtypes can be loaded into Intermezzo at run-time as needed.

User Representation

Another goal of Intermezzo is to provide a consistent, uniform means to access information about users. In the current state of affairs for UNIX applications, the operating system provides some very basic information about the human users of the system: a unique user ID, a login name, home directory location, shell, group membership, and so forth.

Applications often have the need to associate higher-level information or attributes with users however. This is especially true for collaborative applications which, by their very nature, deal with multiple users and must often associate application-specific information with each user.

Intermezzo provides a uniform but extensible user representation which client applications can use to store information about users. Users themselves, through the use of a client application, alter their representations as stored by Intermezzo. These facilities for

This assumption that activity monitoring is necessary for session management, and the foundation of the Intermezzo session management service on activity monitoring have a number of interesting implications.

First, and perhaps most simply, user information is already readily available to applications. Thus, there is no need to code applications to cull user information from across the net in order provide session management. The load on application writers is significantly reduced since they no longer have to produce "who's on" functions.

Second, the activity information can be used more directly to provide *implicit* session management operations. Most current collaborative systems take an approach to session management which is very heavyweight from the human interface perspective. Often there is an initial start-up phase where the initiator lists users (and the hosts where they are expected to reside) with whom collaboration is desired.

A much more natural model is one where collaborators simple open the same document at or about the same time and are thrown into collaboration (or at least are made aware that they are editing the same document, with the potential to collaborate). Because Intermezzo clients have available to them information about not only which users are across the network, but also what applications they are running and which data they are running the application on, this form of very light-weight, implicit session management becomes trivial to implement.

Third and finally, activity information can be used as a controlling input to the session management facilities themselves (see "Expression of Policy and Procedure" below).

So how does the Intermezzo session management service work? As stated before, Intermezzo publishes information about the activity on the network. This published information takes the form of `Activity` objects which can be retrieved and viewed by interested clients with the proper permissions. Each `Activity` object represents a single user working with a single application on a single data set (perhaps a file in the filesystem). `Activity` objects themselves may be thought of as a tuple containing identifiers for the `User`, the `Task` (the actual application being run), the `Object` (the particular thing that task is directed at), and the `Data` (which is used to represent generic stored and shared information made available through Intermezzo for application-specific use).

In Intermezzo, an instance of collaboration (a single, discrete event of collaboration between two or more individuals) is defined as an overlap or confluence in the database of activity information. More concretely, think of the database of activity information as a table in which the rows correspond to individual activities and the columns correspond to the components of an `Activity` object. Intermezzo can detect conditions where two separate rows have the same value in a particular column. As an example, consider Table 1.

Table 1 provides an example where two users are using a total of five applications (Tasks) across the network (for simplicity the table does not show the `Data` component of an `Activity`). Intermezzo can detect a confluence in the activity database. For example, suppose that the `Object` ~/foobar in `Activity` 2 is actually the same file referred to by the `Object` /hm10/keith/foobar in `Activity` 5.

Intermezzo can determine that these two objects are in fact the same and report this information to the client applications (which in this case both happen to be textedit), or even to some other application (perhaps a "collaboration monitor" program). The infor-

about user location or current preferences (“Tom seems to be in a video meeting with his boss so I probably shouldn’t interrupt him”). Having activity information available to applications also means that applications can express policy and preference details based on that information (for example, you may tell your Collaboration Manager application, “never interrupt me when I’m running a spreadsheet on the final quarter earnings report”).

Intermezzo allows applications to specify arbitrarily complex behaviors based on, among other things, the current state of user activity across the entire network.

Performing activity monitoring with Intermezzo requires some very minor changes to applications, but very little “intelligence” on the part of the application developer. Essentially, the application is simply relinked to include the Intermezzo client-side library. With graphical applications, this is most easily accomplished by building a new version of the graphical toolkit library (for example, libxview.so or libXt.so) which itself has the Intermezzo client code linked into it. In such a situation developers don’t even have to be aware that they are participating in the Intermezzo activity monitoring service. By installing the new versions of the shared libraries, all applications built on those libraries automatically participate in the activity monitoring service.

In this usage, applications will “publish” information about themselves across the network. Interested parties can “subscribe” to activity information that they wish to see. At this simplistic level, applications linked with the Intermezzo client library will publish only basic information about themselves and their users. If developers wish, they can directly interact with Intermezzo to tailor the information which they publish (for example, to provide additional useful information, or to restrict access to some information by placing security attributes on it).

It is important to realize however that it is possible for applications to participate in the Intermezzo activity monitoring service without being explicitly rewritten to do so. This was one of the primary pragmatic goals of the system. Of course, developers who wish to can explicitly code to the Intermezzo interface to make better use of the system.

Session Management

Another important service provided by Intermezzo is *session management*. Session management is the process of starting, stopping, and browsing collaborative sessions which are taking place across the network. It involves the process by which users rendezvous together to perform some collaborative activity, how others are made aware of the collaborative event which is taking place, and how those users disengage themselves from collaboration.

It is important to realize the philosophical underpinnings of the Intermezzo session management model. Intermezzo doesn’t make an intrinsic distinction between collaborative and non-collaborative applications. It simply deals with applications. Of course, some applications may make better use of Intermezzo’s features than others (and hence, will be better “citizens” in the collaborative universe).

The Intermezzo session management model is based on the assumption that it is not only important but *necessary* to maintain information about user activity on the network.Thus, the session management facilities in Intermezzo are built on top of the activity monitoring service which is described above. Intermezzo uses the information provided by the activity service to facilitate session management.

Intermezzo provides several facilities which should be useful to collaborative applications. These facilities are designed to enhance the power and interoperability of collaborative systems. Intermezzo provides an activity monitoring service which allows applications (with the proper permissions) to be notified of user activities across the network. Intermezzo also provides support for session management (conference set-up, joining, leaving, and browsing) for collaborative applications. It also provides a convenient rendezvous point for inter-application data sharing. Eventually it will provide facilities to support synchronous and asynchronous collaboration, and transitions between these two modes of collaboration.

Road Map of This Paper

This paper is organized as follows. Section 2, Architectural Overview, provides some high-level details on the implementation of Intermezzo and how its various features are constructed. This discussion is in terms of the goals of the system, such as activity monitoring and session management. Section 3, Shared Data Service, presents an introduction to the capabilities of the layer of software on top of which Intermezzo is built. The Shared Data Service (SDS) provides a general-purpose object storage facility for clients. Section 4, SDS Agent Scripting Interface, delves into the goals and implementation of the SDS scripting interface. Since Intermezzo is built on top of the SDS, these facilities are directly available to Intermezzo clients as well. Section 5, Intermezzo-Specific Resources and Facilities, explains how Intermezzo uses the facilities provided by the SDS to accomplish its work. Section 7, Caveats, provides some details on weaknesses of the current implementation as well as future directions. Finally a number of appendices cover the SDS protocol and the agent scripting language..

Architectural Overview

This section presents a high-level overview of the various architectural features of Intermezzo. Since these features are driven by the set of goals we have in mind for the system, we will discuss the architecture of the system in terms of its goals.

From a high-level perspective, the goals of Intermezzo are: activity monitoring, session management, user representation, data sharing, expression of policy and procedure, and support for multi-modal collaboration. Each of these functions will be addressed in turn.

Activity Monitoring

One of the services provided by Intermezzo is *activity monitoring*. Activity monitoring is the collection of data about current user activities across the network. Activity data has a number of potential uses in a collaborative system. The most obvious is as a basis for providing awareness of user activity. Previous research has shown that awareness of other users is important in a collaborative setting; Intermezzo provides the infrastructure for building awareness of user activity (in addition to whatever other types of awareness information are provided by specific applications, such as video monitoring).

But activity data can be used to a somewhat more subtle end as well. Information about user activities can be used as an input to the system itself (as opposed to simply being used as input to another human being, as in the case of awareness above). For example, a session management service can use information about user activity to “make a guess”

Introduction

A collaboration support environment is a software system which developers can build on top of to provide more robust and flexible collaborative operations. This paper describes the architecture of the Intermezzo collaboration support environment. Intermezzo is a set of libraries, servers, and conventions which developers can use to construct collaborative applications.

Intermezzo does not attempt to provide support for all aspects of collaborative software development. Instead, it restricts itself to providing support for the sharing of "facilitating data." Facilitating data is data which is used by applications and the collaboration support environment itself to facilitate the process of collaboration. An example of facilitating data is information about which users are currently active in collaborations.

Facilitating data is distinct from *artifacts*. Artifacts are the actual data of discourse or concern of the collaborative process. As an example, the text document which is edited by a shared word processor is an artifact. Intermezzo does not provide facilities to support the sharing and use of artifacts.

Figure 1 provides an illustration of a model in which collaborative applications are built on top of two separate toolkits: a toolkit for artifact or application data sharing, and a toolkit for between-application or facilitating information sharing. Intermezzo is the box on the right in the diagram - the facilitating data sharing toolkit. It is expected that to more fully support the development process for collaborative applications, a toolkit whcih supports the sharing of application data (artifacts) would provide services in addition to those provided by Intermezzo.

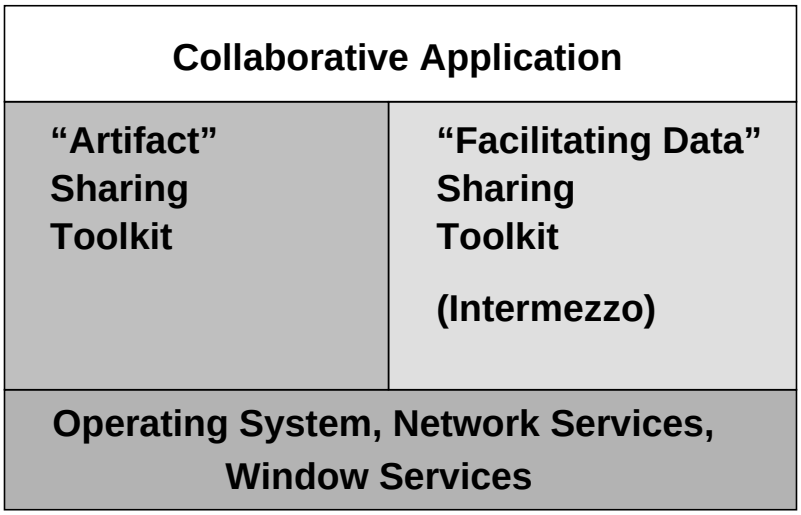

FIGURE 1. Layers of a Collaborative Application

October 20, 1993

Intermezzo Implementation Notes

(version 0.8)

The Intermezzo Collaboration
Support Environment

W. Keith Edwards

keith.edwards@cc.gatech.edu

Graphics, Visualization, & Usability Center
Georgia Institute of Technology
Atlanta, GA 30332-0280

This project is sponsored by Sun Microsystems, Inc.