August 27, 2004

US Department of Energy
SBIR/STTR Program, SC-32
19901 Germantown Road
Germantown, MD 20874-1290

To whom it might concern:

This is the final report for the Tech-X Phase II SBIR project "CORBA for Fourth Generation Languages, "Grant No DE-FG03-00ER83107.

Sincerely,

Svetlana Shasharina
Vice-President of Tech-X Corporation and PI of the project

Table of Contents

Table of Contents 2
Introduction 3
IDL-CORBA Bridge 3
- Architecture 3
- Installation of the Bridge 4
- The IDL Object Interface 5
- IDL CORBA Object Detection 6
- IDL – CORBA - IDL Conversion 6
- Running CORBA Servers 7
- Running IDL Clients 8

CORBA-IDL Bridge 10
- Architecture Overview 10
- Installing CORBA-IDL Bridge 12
- Preparing IDL Objects for CORBA Clients 12
- Running the Bridging Server: 12
- Running the Client 13

TaskDL 13
- Overview 13
- Status of the TaskDL 14
- Tuple Space 15
- Task Tickets 16
- Directory Structure 16
- Graphical User Interface Description 18
- Session Control 18
- Session Configuration 20
- Worker Output 21
- Session Status 22
- Worker Status 22
- Task Status 22
- IDL Procedure Signatures 23
- The Setup Wizard 23
- Installation 25
- Running TaskManager from CVS 25

Introduction

The software developed in this project consists of three parts: ILD-CORBA bridge, CORBA-IDL bridge and TASKDL. In what follows we discuss these pieces along with instructions on how download and use them. IDL-CORBA bridge allows IDL code to access remote objects written in other languages. CORBA-IDL bridge allows using remote IDL server objects in client codes written in other languages. Finally, TASKDL is an attempt to create a distributed client-server using light-weight approach alternative to CORBA and run parallel IDL applications.

IDL-CORBA Bridge

Architecture

The Common Object Request Broker Architecture (CORB is an open distributed object infrastructure defined by the Object Management Group (OMG). OMG is an industrial consortium that, among other things, oversees the development and evolution of CORBA standards and their related service standards through a formal adoption process. The process has proven to be highly effective in ensuring the quality, the interoperability, and the implementability of newly adopted standards. CORBA standardizes and automates many common network programming tasks such as object implementation, registration, and location transparency. CORBA also defines standard language mappings of most popular languages for the programming interfaces to services provided by the Object Request Broker (ORB). An ORB is the basic mechanism by which objects transparently make requests to and receive responses from other objects on the same machine or across a network.

By ensuring the quality and the interoperability of the standards, ORB venders are able to continuously evolve and optimize ORB implementations and users can freely switch between ORB products with minimal efforts. The General Inter-ORB Protocol (GIOP) is CORBA's standard message exchange protocol over the wire. The Internet Inter-ORB Protocol (IIOP) is a concrete realization of GIOP using TCP/IP. GIOP adopts a binary data representation format called the Common Data Representation (CDR) that allows standardized and efficient message exchange between ORBs. Unlike Web Services which use XML documents as the common format for message exchange, CORBA uses CDR. This allows messages to be exchanged in binary format and is much more efficient both in terms of processing overhead and protocol overhead.

Current implementations of CORBA do not provide bindings for Interactive Data Language (IDL). Our IDL-CORBA-Bridge partially gaps this problem by providing capabilities to call CORBA objects from IDL client. As a consequence, scientists and engineers will be able to create IDL client applications to access remote CORBA objects and use the powerful features of IDL to analyze or render that data locally.
Our bridge uses TAO, an ORB developed by the Distributed Object Computing Group of Washington University in St. Louis, the University of California, Irvine, and Vanderbilt University. TAO is an open-source, high-performance and highly configurable ORB implementation of CORBA specifications. TAO supports the standard OMG CORBA reference

model and real-time CORBA specification with many enhancements designed to ensure efficient, predictable, and scalable QoS behavior for high-performance and real-time applications.

The IDL-CORBA bridge uses C++ implementation of Dynamic CORBA and IDL Dynamic Loadable Modules wrapping Dynamic CORBA, to provide the connection between IDL client application and CORBA objects. The current implementation of the bridge requires a C++ compiler (gcc3.3 and higher), TAO (compiled with corresponding gcc) and IDL (version 6.0 or higher). It works on Linux and Unix platforms.

The general architecture of the IDL to CORBA Bridge is shown on Fig.1. As indicated in this diagram, the architecture consists of four main subsystems: the IDL object interface, the "tap" into the IDL object system, the conversion between IDL and CORBA primitives and CORBA itself.

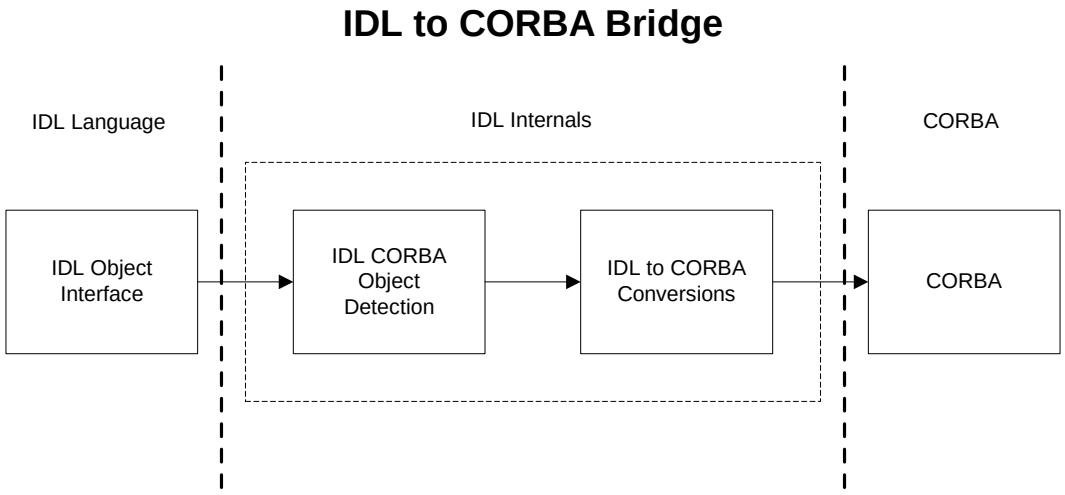

Fig. 1. Basic architecture of the IDL-CORBA bridge.

As indicated in this diagram, the architecture consists of four main subsystems: the IDL object interface, the "tap" into the IDL object system, the conversion between IDL and CORBA primitives and CORBA itself. The specifics of each system is discussed in the following sections.

Installation of the Bridge

Unpacking the tarball with the command

```
gunzip –c idlcorba.tar.gz | tar xfpv –
```

will create idlcorba directory with the following subdirectories:

- idl_corba_bridge - the main directory containing the bridge source code;
- server_cpp - a directory containing a demo server object, the test C++ server code using this demo object and scripts needed to run the server side of the bridge;
- client_idl has several examples of IDL client using test objects from server_cpp directory;

- stddir - a utility directory;
- config – directory with files included in configure.in and needed to generate Makefiles.

The package comes with all items needed to generate Makefiles. In order to create configure the first time, one needs to run scripts

```
./config/cleanconfig.sh
```

```
./config/regenconf.sh
```

In order to make Makefiles, run

```
configure
```

Then do

```
make depend
```

```
make
```

To clean (delete generated files), do

```
make clean
```

This implied that you autoconf and configure tools installed. The default implies that the main TAO installation directory is /usr/local/ACE+TAO. If it is different, configure script should be run as follows:

```
configure  --with-corba-idl=<full path of tao_idl>
```

For example:

```
configure –with-corba-idl=/usr/local/installations/ACE+TAO/TAO/TAO_IDL/tao_idl
```

If location of IDL different from /usr/local/rsi/idl, one needs to run configure as follows

```
configure –with-rsiidl-bindir=<location of IDL bin directory>
```

For example,

```
confgure –with-rsiidl-bindir=/usr/local/idl/bin
```

Note that you can run configure with multiple concatenated "—with-" statements.

The IDL Object Interface

As discussed earlier, CORBA objects are exposed at the IDL language level as normal IDL objects. As such the user performs standard IDL operations and syntax to use these proxy objects. All the obj_* routines exposed in IDL operate on a CORBA object and method calls are initiated using the standard IDL method operator, `->`. While the standard IDL object operations are used, some unique methodologies are required to create the link between IDL and the underlying CORBA object. This interface was provided to Tech-X by RSI.

CORBA objects are defined in .idl files (IDL here stands for Interface Definition language, which we will refer to as CORBA-IDL). These files show interfaces of the objects. For example, this snippet of the code defines interface Demo:

```
// Demo.idl
interface Demo {
        long doLong(in long li, out long lo, inout long lio);
```

```
};
```

with one method, which return a long, takes li as an input parameter, lo as an output parameters, lio as both input and output parameter.

CORBA-IDL compiler takes this code and creates skeletons and stubs. In case of TAO, this is done using the following command:

```
/usr/local/ACE+TAO/TAO/TAO_IDL/tao_idl -Ge 1 -Sc -Ge 1 -Sc Demo.idl
```

assuming that ACE_TAO is installed in /usr/local.

Skeletons are abstract classes, which are used on the server side. For example, interface Demo, will obtain a skeleton class POA_Demo. Server implementation class should inherit from the skeleton and implement methods declared in the .idl file. Here is an example of a C++ implementation (see DemoImpl.h and DemoImpl.cpp):

```
class DemoImpl : public POA_Demo {
 public:
      CORBA::Long DemoImpl::doLong(CORBA::Long li,
                                   CORBA::Long& lo,
                                   CORBA::Long& lio) throw(CORBA::SystemException) {
      TXSTD::cout << "doLong" << TXSTD::endl;
      lo = li + 1;
      lio += li;
      return lo;
}
```

Directory server_cpp contains an extensive example using all types supported by the bridge. In order to generate stub and skeletons, one needs just to do “make depend.” In addition, this directory contains the implementation (DemoImpl.h and DemoImp.cpp), which can be compiled by doing “make.”

IDL CORBA Object Detection

The next functional area of the IDL to CORBA bridge architecture is the detection of IDL operations on CORBA specific objects. This area involves the addition of logic to the core IDL implementation that allows for the detection of IDL CORBA proxy objects during IDL object creation and method calls. During these operations, if an IDL CORBA proxy object is detected, operational control is shifted to the IDL CORBA system. No CORBA specific technologies or interfaces are utilized in this functional area of the architecture. This part of the bridge was also implemented by RSI.

IDL – CORBA - IDL Conversion

The IDL-CORBA bridge supports a rich subset of variables. Here is the corresponding mapping table of supported types.

CORBA Type	IDL Type
short	IDL_INT
unsigned short	IDL_UINT

long	IDL_LONG
unsigned long	IDL_ULONG
float	float
double	double
char	UCHAR
boolean	UCHAR
octet	UCHAR
string	IDL_STRING
Arrays of the above	Arrays of the above

CORBA operations parameters have directional attributes:

- `in` – The `in` attribute indicates that the parameter is sent from the client to the server and can be used, but not modified by the function call.
- `out` – The out attribute indicates that the parameter is sent from the server to the client. Its value will be set by the server function call.
- `inout` – The `inout` attribute indicates that the parameter is first sent by the client to the server and then back to the client. Its value maybe reset by the server function call.

This means that the IDL client code should always instantiate `in` and `inout` parameters. It may instantiate `out` parameters, but their values will be set in the process of invoking the server function.

This architectural area comprises the majority of the functionality required to implement the IDL to CORBA Bridge. In this system, requests are validated, IDL data is converted into a format recognized by CORBA and method calls are packaged and dispatched to the target CORBA object. All of these operations are performed dynamically to provide the runtime behavior expected by the IDL user.

The key technology that allows the dynamic method dispatch that this subsystem performs is the CORBA Dynamic Invocation Interface (DII). This interface provides the ability to call methods on CORBA objects without specific object information during system implementation. When a method is called on an IDL CORBA proxy object, the following procedure takes place:

1. IDL determines if the method exists on the target object.
2. IDL parameters are packaged as CORBA Any variables.
3. The method call is dispatched to CORBA using the functionality provided by the DII system.
4. Upon return from the method call, any output values are retrieved, converted back into IDL variables and control is return to IDL.

These conversions were implemented by Tech-X and comprised the biggest part of the bridge implementation.

Running CORBA Servers

In order to access CORBA objects, one needs to instantiate the ORB and the objects themselves. An example can be found in server_cpp directory: files serverIOR.cxx and serverNS.cxx. The

difference between these server executables is that the CORBA object publishes its reference differently.
In the first case (serverIOR.cxx), the ORB creates an IOR (Interoperable Object Reference), which encapsulate the server port number, its IP address, transport and protocol used and the name of the object. In our example, this IOR is written to a file Demo.ref. This file should be available to the client. If one needs to access many CORBA objects, using IOR is not practical. It makes more sense to register all objects with one process and access them through this process using objects names. This is achieved by using the CORBA Naming Service (see serverNS.cxx). In case of TAO, one can start it by the following command:
In the second case, the CORBA object registers itself with the Naming Service, which allows obtaining objects references from this service by objects names. In order to use Naming Service, one needs to start it by starting the daemon before starting the server process. For example:

```
/usr/local/ACE+TAO/TAO/orbsvcs/Naming_Service/Naming_Service -ORBEndpoint
iiop://`uname -n`:50065/ &
```

where 50065 is the Naming Service port number.
The bridge extensively uses Dynamic Invocation Interface (DII) for extracting knowledge about an OMG-IDL interface. In order to register interfaces, one needs to start the Interface Repository and populate it with .idl files. In case of TAO, this is done as follows:

```
/usr/local/ACE+TAO/TAO/orbsvcs/IFR_Service/IFR_Service -ORBEndPoint iiop://`uname -
n`:50063 &
/usr/local/ACE+TAO/TAO/orbsvcs/IFR_Service/tao_ifr -ORBInitRef
InterfaceRepository=corbaloc:iiop:`uname -n`:50063/InterfaceRepository Demo.idl &
```

where 50063 is the port number of the repository process.
After all the needed processes (the Interface Repository for both cases and the Naming Service for the second case) are started, one then can run executables serverIOR and serverNS, respectively.
All the steps described above are collected in two shell scripts: run_serverIOR.sh and run_serverNS.sh. One should edit them to change the default port numbers and names of .idl files used. Note that these numbers as well as the server IP name should be available to the client code.

Running IDL Clients

IDL client can access server objects (after servers are started as described above) via proxy IDL objects. These objects are created with the OBJ_NEW function with a specific CORBA token. This token makes sure that when a function or procedure is called using the proxy object, all methods are delegated to the bridge. The bridge parses the object name and the method signature, converts IDL variables into CORBA variables, creates a Dynamic CORBA request and sends the request to the servant object. When the request returns, the bridge converts CORBA variables back into IDL variables and makes them (including the return value) available to the IDL application. The OBJ_DESTROY destroys the proxy object and releases resources associated with CORBA.

Corresponding to the two types of the servers, there can be two kinds of IDL clients: using IOR and the Naming Service. Examples are given by idl_clientIOR.idl and idl_clientNS.idl in the

client_idl directory. They use correspondingly serverIOR and serverNS processes. Note, that idl_clientIOR.idl counts on Demo.ref file, containing IOR of the Demo object, to be in the same directory. One can run getIOR.sh script to get the file (edit according to where Demo.ref file was generated) or modify idl_clientIOR.pro to refer to correct location of this file. Also, note that the client code uses DLM's which are located in the idl_corba_bridge directory, but the make process should have created a link to the required shared library (idl_objbridge_corba.so) in the client_idl directory.

To run idl_clientIIOR, one needs to pass information about the Interface Repository. For example, if the repository is running on quad.txcorp.com on port 50063, one needs to do the following:

```
IDL> .r idl_clientIOR
% Compiled module: IDL_CLIENT_IOR.
IDL> idl_clientIOR, "quad.txcorp.com:50063"
```

To run the client using the Naming Service, one needs to pass information about this process too:

```
IDL> .r idl_clientNS
% Compiled module: IDL_CLIENTNS.
IDL> idl_clientNS, "quad.txcorp.com:50063", "quad.txcorp.com:50065"
```

where the processes port numbers should correspond to numbers used in the server scripts run_serverIOR.sh and run_serverNS.sh. If everything is OK, you will see the following output:

```
doDouble
f should be 3, it is       3.0000000
x should be 1, it is       1.0000000
y should be 3, it is       3.0000000
z should be 3., is is       3.0000000
doLong
f should be 2, it is           2
x should be 1, it is           1
y should be 4, it is           4
z should be 2, is is           2
doFloat
f should be 6, it is      6.00000
x should be 3, it is      3.00000
y should be 9, it is      9.00000
z should be 6, is is      6.00000
doShort
f should be -4, it is      -4
x should be -3, it is      -3
```

```
y should be -5, it is       -5
z should be -4, is is       -4
doUShort
f should be 3, it is        3
x should be 3, it is        3
y should be 5, it is        5
z should be 4, is is        4
doULong
f should be 4, it is            4
x should be 3, it is            3
y should be 6, it is            6
z should be 4, is is            4
doString
f should be hello, it is hello
s1 should be hello, it is hello
s2 should be hello, it is hello
s3 should be hihihi, is is hihihi
doVoidParam
f should be 2, it is            2
doReturnVoid
q should be 2, it is            2
doOctet
f should be B, it is B
o1 should be A, it is A
o2 should be B, it is B
o3 should be C, is is C
fa should be [4, 1, 5], it is          4          1          5
la1 shoudl be [4, 1, 5], it is          4          1          5
la2 should be [5, 2, 6], it is          5          2          6
la3 shoudl be [14, 2, 6], it is         14          2          6
```

CORBA-IDL Bridge

Architecture Overview

Similarly to our approach in implementing the IDL-CORBA bridge, we decided to use Dynamic CORBA for the CORBA-IDL bridge design, namely the Dynamic Skeleton Interface (DSI).

This approach allows disengaging the server implementation from the OMG-IDL interfaces and skeletons generated from them by CORBA compilers.

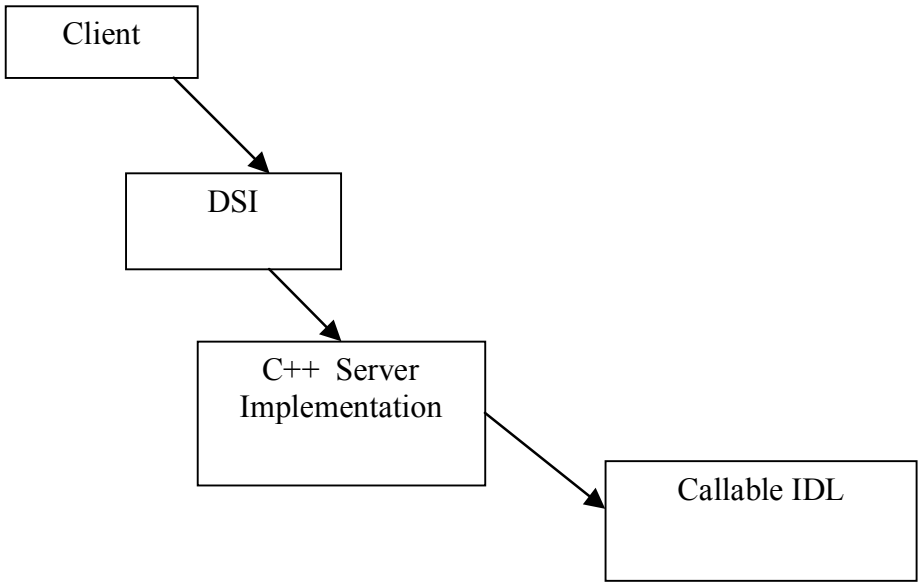

Figure 2. High-level architecture of the CORBA-IDL bridge.

In this approach the DSI servant plays the role of adapter: it receives CORBA invocations from the CORBA client, translates them into dynamic language of a C++ dynamic server, which is implemented to use Callable IDL, thus delegating the work to IDL objects.

Several C++ classes were created to implement this architecture. Class TxRsiCorbaObjectsBridge is the DSI servant, which is created by the server main program and is derived from the DynamicImplementation class. This object starts a single IDL process, represented by a singleton TxCallableIDL class, whose constructor calls IDL_INIT and who creates IDL objects registered with the bridge. It also creates a TxRsiObject, which represents the information obtained via interaction with the Interface Repository: practically a list of TxOperations of each interface, thus representing remote IDL objects in the bridge.

The main task of DSI implementation is to implement the invoke() method of the servant (here TxTsiCorbaObjectsBridge). The following things happen in our implementation of this function. First, the identity of the requested object is determined and checked with the objects available in the registered list. The name of the operation is determined next. This allows us to find the list of parameters of the operation, create an NVList and package the request object with correct arguments. After that the action is passed to the TxRsiObject's handleOperation() function.

This function tells the correct operation to handle itself: which results in building a correct string and data structures needed for Callable IDL. After that the static singleton TxCallableIDL takes the description string and data structures and finish the work.

Installing CORBA-IDL Bridge

CORBA-IDL bridge is distributed in a file called "corbaidl.tar.gz". Please refer to "Installation of the Bridge" section of IDL-CORBA bridge for details instructions on how to unpack, configure, and build the CORBA-IDL bridge. Unpacking the distribution will create the following major subdirectories:

- `corba_idl_bridge` is the main directory containing the libraries for the bridge between CORBA clients and IDL implemnetation.
- `server_idl` contains the generic the C++ server code for IDL objects and scripts to start up the IFR service.
- `client_cpp` contains an examples CORBA client for invoking an IDL object.

Preparing IDL Objects for CORBA Clients

Files containing RSI IDL object implementation should be named according to IDL convention. For example, our Demo object is defined by the file "Demo__define.pro". Once you have the IDL object implementation ready, you need to decide what functions/procedures of the object need to expose to CORBA client, you need to define the interface for CORBA client using the CORBA interface definition language (CORBA IDL.) An example CORBA interface definition is available in `client_cpp/Demo.idl`.

A few hints to keep in mind when creating the CORBA interface defintions:

- IDL functions have return values while procedures have not. Map IDL procedures to CORBA functions with void return type.
- Be careful to assign argument passing direction (in, out and inout).
- Many RSI IDL functions can be used with many different data types (e.g., integer, double, array of integers and array of doubles) safely without modification. However, functions defined in CORBA interface can only be used with specific types. You will need to create adapter RSI IDL functions with different names if you wish to use the same function with different data types.

Running the Bridging Server:

First you need to

```
cd server_idl
```

In order to support arbitrary CORBA interfaces in a single server, the server needs to query the CORBA Interface Repository to acquire the definition of the CORBA interface during execution. Therefore, before running the server, we need to have the Interface Repository running and

initialize with all the interface definitions the server may need to serve. The `start_ifr.sh` shell script under the `server_cpp` subdirectory shows how to start up the TAO Interface Repository service and how to feed the repository with the client side CORBA interface definition. `start_ifr.sh` also shows how to feed the example Demo.idl defined under client_cpp subdirectory to the Interface Repository. Notice that you may need to update the path inside the script about where TAO's interface repository service program can be located.
Once the Interface Repository is started, you can start the server as shown in `run_server.sh` shell script. By default, the server implementation will be initialized to service the "Demo" object defined in this directory. The generated IOR key that identify the object will be written to a file called "Demo.ref" which the client will use to get access to the RSI IDL object. Users can override:

- The name of the RSI IDL object to be served in this server by using the `-i <name>` command line flag. Notice that there must be a corresponding `<name>__define.pro` available.
- The name of the file the IOR will be written to by using the `-o <ior filename>` command line flag.

Running the Client

This `client_cpp` subdirectory contains an example of how to invoke functions defined in an RSI IDL objects. Usually, the IDL object developers define the CORBA interface clients can access the object. The `Demo.idl` file provides one such example interface definition for the example IDL object defined in the `../server_idl/` subdirectory.
After starting up the server in `../server_idl/` subdirectory, you can start up the client as:

```
$ ./client -k file://../server_idl/Demo.ref
```

The only command line flag `-k` is used to passed in the Interoperable Object Refernce (IOR) that uniquely identify the instance of RSI IDL object the client should contact.

TaskDL

Overview

The goal of TaskDL (http://grid.txcorp.com/taskdl/index.jsp) is to provide an environment to easily set up and run IDL applications on parallel, distributed grid-like WAN resources for faster and more efficient execution. TaskDL is an effective tool for increasing performance when the underlying parallelized tasks require no communication with each other.

For instance, rendering frames for a movie or processing spatial data from a series of time slices represent problems, which are parallelizable in a task-farm paradigm such as TaskDL. The TaskDL Task Manager sets up a farm of parallel host workers which process tasks

independently, achieving excellent speed-up as a function of number of worker nodes. TaskDL provides a simple user interface and requires very little refactoring of existing serial code. TaskDL is a component of the FastDL umbrella of products. For more information, contact info@txcorp.com.

QuickTime™ and a
TIFF (Uncompressed) decompressor
are needed to see this picture.

Status of the TaskDL

A user may federate IDL hosts without installing the TaskDL software on each of them. TaskDL requires that the host workers have a common NFS space with each other and the session manager host so that all processors can access the same data, log, and session directories (the tuple space). The cluster of IDL worker hosts must each have a licensed installation of IDL.

The manager application starts the host workers with SSH. TaskDL supports up to two separate connections to each worker node in case worker hosts have dual processors. Coordination between the manager application and host workers is done only through the tuple space populated with work tickets.

Tuple Space

The concept of a Tuple Space has been developed as an underlying paradigm for distributed and parallel computing using shared memory. A tuple is a vector of values, which describe an object, such as a data file, resource, process or protocol. Distributed processors can discover work to be done by looking for structured XML task tickets in the Tuple Space, and can communicate with each other by moving tuples within the space, removing tuples from the space altogether, and writing new tuples into the space. See xml.coverpages.org/tupleSpaces.html for some examples of how Tuple Spaces have been developed for distributed and parallel programming

The tuple space used by TaskDL is a filesystem containing a collection of work tickets, log files, and data files accessible by the worker nodes. Work tickets are tuples containing XML tagged descriptors of data, IDL procedure names, which will process the data, and the locations of data and IDL code needed to complete the parallel task. The tuple space is implemented by creating task tickets in XML and moving them between directories that denote their status. The manager creates task tickets and puts them into a ***to-do*** directory. When a worker picks up a task, it moves the ticket to a ***work-in-progress*** directory, examines the XML to see how to do the work, and processes the input data by running the IDL procedures specified in the task ticket. File system locks provide synchronization between workers so that they don't try to process the same task. When the worker is done with a particular task, the ticket is moved to a ***done*** directory and the ***to-do*** directory is re-examined for more work tickets. The status of the tasks and the workers are written to log files and updated in the TaskDL application GUI. When the list of tasks in the ***to-do*** directory is exhausted, the workers exit IDL. The last remaining worker processes the serial finalization task (if there is one) and then exits IDL.

If a task can not be completed for some reason, such as a worker node failure, an unavailable IDL license, or a corrupted input data file, the user has the ability to manually re-queue the task so that a different worker can pick up the ticket and complete the task. Once the manager application has started the workers, they operate independently, and the manager application can be stopped without affecting the progress of the task farm. At some later time, the user may reconnect to an existing session and determine the progress of the tasks. The reconnection may be made to tasks that are still actively being processed as well as completed tasks.

An example of how the TaskDL system works is creating a movie from a large set of time-sliced two-dimensional data. In a movie farm, ***data.xml*** tickets represent the data and IDL code which will produce individual frames of the movie through some image analysis and production procedure, creating image files (such as PNG). The finalization step would combine the resulting images into a movie file, such as MPEG. The worker nodes will process individual frames in a parallel manner, as there is no dependencies between the frames. The finalization procedure can not be processed until all of the frames have been produced, and must be done in serial.

Below is an outline of how the tuple space is implemented in TaskDL. Not shown are the data and code tuple spaces. Task tickets are moved to directories, which represent the state of the

task. In addition, log files are maintained and are used by the manager to display the session status in the GUI.

Task Tickets

The XML describing a task ticket is fairly basic. Below is an example ticket, showing the paths to the data and parallel procedure to be run by the workers are given. The XML task tickets are generated by the manager application when the session is started. They are parsed by the IDL workers. The function calls to the parallel work procedures in IDL with the correct input data files for each task are generated using the parsed XML variables.

```
<task>
<task_id>task1</task_id>
<data_loc_array>
<data_loc>/Users/veitzer/data/d1.dat</data_loc>
<data_loc>/Users/veitzer/data/d2.dat</data_loc>
</data_loc_array>
<pro_loc>/Users/veitzer/mycode/plex.pro</pro_loc>
<procedure>plex</procedure>
</task>
```

Directory Structure

The directories generated by TaskDL implement the tuple space, which must be cross-mounted on the manager host and all worker nodes. The session_<SESSION-ID> active directory is automatically created based on the system time when a session is configured and executed within the GUI. When all work for the session is completed, the entire session directory name is changed, indicating not only the time that the session began, but also the time when the session was completed and the final state of the session (completed or completed-with-failures). The tuple space for a given session is under the session directory tree. Note that the input, output, and user-provided code directories must exist outside of the session directory because they persist independently of the session being run.

```
/shared
  /examples
```

```
  /movie
    /code
      create_png.pro
      create_mpeg.pro
    /input
      *.dat
    /output
      *.png
      movie.mpeg
  /montecarlo
    /code
      calcpi.pro
      average.pro
    /input
      pi_*.in
    /output
      pi_*.out
/txcode
  catch_idl_lm_error.pro
  drop_ticket.pro
  parse_file.pro
  task_parser__define.pro
  worker.pro
/session_<SESSION-ID>.active
  /todo
    task*.xml
  /wip
    task*.xml
  /done
    task*.xml
  /state
    manager.state
    manager.tasks
    manager.workers
    host*.state
  /logs
```

```
host*.stderr
host*.stdout
```

For more general problems, the user will define their own cross-mounted input, output, and IDL code tuple space to be used instead of the examples. These directories can be specified in the GUI.

```
/input_data
  *.dat
/output_data
  *.out
/user_code
  unit.pro
  final.pro
```

Graphical User Interface Description

This section describes TaskDL's user interface. When the TaskDL application is started, the TaskDL Task Manager GUI is brought up with the last configuration loaded. The Manager provides information about the current session, such as the status of worker nodes and tasks, output from the IDL workers, progress of the session, and the general configuration of the tuple space. The Manager also provides session control through buttons which allow the user to create a new configuration, load/save configurations, execute a session with the current configuration, reconnect to a previously executed session, check the status of host workers and possibly requeue their tasks, and quit the application. Note that once a session is started, it runs independently on the remote worker nodes, whether or not the Manager is connected to the session.

Session Control

This panel contains information about the Manager and the session timing, as well as buttons which control the session and the GUI. This control is opened when the application starts.

- Displayed Information
  - Manager Host - The name of the host machine which is running the session manager and hence the GUI itself.

  Note: The manager host must be cross-mounted with all of the worker nodes.

  - Session Started - Displays the date and time that a session is first executed.

- Session Ended - Displays the date and time that a session finishes.

- Buttons
  - Execute - This button begins execution of a session using the current session configuration (see Session Configuration below). Pressing the EXECUTE button starts a number of processes behind the scenes, such as creating session-specific directories and files, generating xml descriptions of tasks to be processed, generates IDL batch files and shell scripts, and writing state and configuration files. The EXECUTE button also establishes ssh connections to the worker nodes.

  - Reconnect - This button allows the manager to reconnect to a session, which was previously started. Once the manager has made ssh connections to the worker nodes and started the IDL processes, the workers act independently without any control from the manager. The manager reports the progress of the workers in the GUI, but may be stopped without affecting the worker progress. If the manager is restarted at a later time, using the RECONNECT button allows the manager to catch up with the progress of the workers and display the current state of the session in the GUI. It is possible to reconnect to both active sessions (where work is still being done) and also to completed sessions (where all work has ceased.)

  - Worker Status - In the event that a worker fails while executing a task, it may be necessary to manually rqueue that task so that it may be processed by another worker. The WORKER STATUS button provides an interface for the user to get detailed information on the status of a given worker. Choosing the WORKER STATUS button first determines the last reported state of the worker, returning the name and status of the worker host, the last task that the worker was working on, and the last reported status of the task. If the status of the last task is still active, it may be that the worker has in some way failed, and the task is not going to be completed. In this case the manager prompts the user to reconnect to the worker host via a new ssh connection, and check to see if the IDL process is still running on the worker. If either the ssh connection can not be made, or if the IDL process is not longer in the process table, the user may requeue the task so that a different worker may process the task at a later time. If the user chooses to requeue a task, it is assumed that the worker host is dead and will not process any further tasks.

  - Quit - Quit the session manager application. Quitting the manager does not affect the progress of any of the worker hosts.

Session Configuration

The SESSION CONFIGURATION panel displays the user-defined parameters for the session. These parameters may change from session to session, depending on the needs of the user. The panel also contains buttons for creating, loading, and saving session configurations.

- Displayed Information
  - Input Directory - The full path name of the directory, which contains all of the input files. The input files are generally data files, which are passed to the user defined IDL procedure which is being run in parallel. For example, if the IDL work procedure reads in a set of data files called frameXX.dat, where XX = 1,2,..., then those files must be located in the input directory. No data is written to the input directory, and no files are edited or removed from the input directory. The input directory should only contain data files, which are to be processed, ie. no description or readme files.

  - Ouput Directory - The full path name of the directory which any generated output files will be written to. For example, if the IDL work procedure processes each frame of data from the files frameXX.dat and writes out a new data file called resultXX.dat, then those output files will be created in the output directory. If the specified output directory does not exist when the session is started, it will be created.

  - Work Procedure - The full path name of the IDL .pro file which contains the user-defined procedure which is to be run in parallel. Note that this parameter specifies the name of the file, not the name of the procedure. If the file specified here contains more than one valid IDL procedure, the user will be prompted to choose the appropriate work procedure. However, all IDL procedures and functions in the file specified here are compiled by IDL, so if the parallel work procedure makes function calls to other user-defined procedures or functions (such as to set up data structures or initialize variables) they should be put in the same file as the work procedure. The IDL work procedure must conform to a specific signature in terms of inputs to the procedure (see section below regarding IDL procedure signatures). Any data files, which are passed to the work procedure must be located in the input directory.

  - Finalization Procedure - The full path name of the IDL .pro file which contains the user defined procedure which is to be run in serial after all of the parallel work has be completed. Note that this parameter specifies the name of the file which contains the finalization procedure, not the name of the procedure itself. However, it is highly recommended that the name of the file and the name of the procedure be the same. All procedures and functions in the specified file are compiled by IDL, but the user will not be prompted to choose the procedure, which is to be run as the finalization code. The finalization procedure is useful when the user wants to run additional co,de which takes data generated by the parallel work procedure as input. For instance, if the work procedure produces output data resultXX.dat, and these data files are to be put together (say making a movie from a series of frames) then

the finalization procedure can accomplish this by serially processing the data files produced by the parallel work processing. Any data files that are used by the finalization procedure must be located in the output directory, including files generated by the work procedure. There are specific signature requirements for the finalization procedure. See below. It is possible to specify that no serial finalization should be performed. If this is the case, then this field will display "<none>".

- Buttons
  - New Config - The NEW CONFIG button launches a setup wizard which allows the user to enter new parameters for running a new session. See the section below on the Setup Wizard for detailed documentation. The user should use the NEW CONFIG button to change session variables, like the output directory or the number of worker hosts.

  - Load Config - The LOAD CONFIG button loads a configuration, which has been previously saved by the user. The user may browse the directory tree to find saved TaskDL configuration files. Configuration files have the extension .tdl.

  - Save Config - The SAVE CONFIG button saves the current configuration to a user named .tdl configuration file. This file may be loaded later on in a different session to restore the current session configuration. Note that the SAVE CONFIG button does not save any information about the current session, only information about the configuration of the session prior to running the session. Information about a running session is automatically saved, and can be accessed at a later time by using the RECONNECT button in the Session Control panel.

TaskDL configurations are saved in two files; a configName.tdl file and a configName.workers file. These files are paired, and are both required when restoring a particular configuration. TaskDL configuration files contain the following information:

- A date stamp indicating when the configuration files were created.
- The full path name of the Input Directory.
- The full path name of the Output Directory.
- The full path name of the file which contains the parallel work procedure.
- The full path name of the file which contains the serial finalization procedure.
- The full path name of the base TaskDL package directory (the location where ant is run).
- The username of the user running TaskDL.
- The full path name of the user's ssh known_hosts file (typically ~user/.ssh/known_hosts).
- A list of worker host names, including information about which connection is specified (a single host can have up to two different ssh connections).
- A list of the locations of the IDL binary for each worker host.

Worker Output

The WORKER OUTPUT panel contains panels with tabs which show informational messages from each IDL process which is running on the worker hosts. The user can switch between panes by clicking on the tab corresponding to the worker, which you would like to observe. In fact, the workers do not write to the tabs themselves, but rather they redirect their output to various log files, which are parsed by threads in the manager. Because of this, the manager can be disconnected, and upon reconnection the worker output panes will catch up to the current state of each worker.

Session Status

The SESSION STATUS panel provides information about the current state of the session. The status bar will indicate some of the key aspects of the session status, such as "connecting to workers," and will report the number of tasks completed. As the session finishes, the total amount of time to complete the session is also reported in the SESSION STATUS panel.

Worker Status

The WORKER STATUS panel provides information on the status of the host workers. The panel displays a table with rows for each host (including connection indicator), the current status of the host, and the Process ID for IDL running on that host. Possible states for the hosts are:

- idle - The manager has not initiated a ssh connection to this host.
- login cancelled - ssh login to this host was cancelled. The host is not doing any processing.
- login failed - ssh login to this host failed. The host is not doing any processing.
- no IDL license - There is a problem with the IDL license manager and IDL can not be run. The host is not doing any processing.
- working - The host is currently running IDL and processing tasks.
- done - There are no more tasks for the host to process. IDL has finished and exited.

Task Status

The TASK STATUS panel displays information regarding the status of all of the tasks in the session. When a new session is executed, the manager creates an XML descriptor of each input file, and puts that 'ticket' into the tuple space to be processed by the worker nodes. The manager then populates those tickets into the TASK STATUS panel. Note that the name of the task is the same as the name of the data file which is to processed. In addition, if there is a finalization task, then that is added to the table as well, although there is no XML ticket for finalization

procedures. When worker nodes process a task, they move the XML ticket out of the 'todo' tuple space into the 'work in progress' space, and eventually move the ticket to the 'done' tuple space. In this manner each task is only processed by a single worker. As a worker processes a task, the status of the task is updated by the manager, and displayed in the TASK STATUS panel. The possible task states are:

- queued - The XML ticket for the task in question is in the 'todo' tuple space, waiting to be processed by a worker node.

- initiated - The XML ticket for the task in question is in the 'work in progress' tuple space, and the worker is currently processing the task in IDL.

- completed - The XML ticket for the task in question is in the 'done' tuple space, and the worker has completed the task and written out any output data.

- requeued - A worker initiated the task and then for some reason stopped working on the task. The user chose to move the task from the 'work in progress' space back to the 'todo' tuple space so that a different worker could process the task at a later time. See the section on the WORKER STATUS button in the SESSION CONTROL panel.

IDL Procedure Signatures

TaskDL host workers support only certain specific signatures for the parallel work procedure and serial finalization procedure. Namely, the parallel work procedure must take as an input argument a single name of a data file to be processed. The parallel work procedure cannot return any values, but may directly write out as many output files as is desired. It is discouraged behavior to provide absolute path names for any output files, although this is supported. Output files will by default be created in the specified output directory, which is required if the finalization procedure needs to use these files as input.

The finalization procedure can take no arguments as input parameters. However, the user can put in required input files by hardcoding the names of the files or by using built-in IDL functionality to do file searching and listing, such as the function FILE_SEARCH, to specify the input files for the finalization procedure. See the examples provided with the TaskDL distribution for a simple method for doing this. The finalization procedure can not return any values, but may directly write out as many output files as is desired by the user. All output files from the finalization procedure will be written to the user specified output directory, unless hard-coded paths are provided.

The Setup Wizard

The SETUP WIZARD is a utility that provides a way for the user to specify the parameters of a TaskDL session. There are five panels of the SETUP WIZARD, corresponding to all of the session parameters.

1. INPUT DIRECTORY - Enter the full path name of the directory which contains the input data files for the parallel work procedure. This directory should *only* contain input files (and possibly other directories). Any regular files in this directory will are assumed to be valid input files to the parallel work procedure. Subdirectories are not searched for input files (try using soft links if you want to maintain separation of file locations, although this has not been tested.) Files in this directory are not moved or modified by TaskDL. The user may use the "browse" button to specify the INPUT DIRECTORY location.

2. PARALLEL IDL PROCEDURE FILE - Enter the full path name of the IDL .pro file, which contains the procedure which will work in parallel to process the input files. Note that this input parameter specifies the name of the file. which contains the procedure, not the name of the procedure, although these may be the same. If there is more than one valid IDL procedure in the file, the user will be prompted to specify which procedure is the appropriate one. The user may use the "browse" button to specify the PARALLEL IDL PROCEDURE FILE.

3. FINALIZATION PROCEDURE - Enter the full path name of the IDL .pro file which contains the serial finalization procedure. Note that this input parameter specifies the name of the file, not the name of the procedure. However, they should be the same, and the user is not prompted to specify the name of the procedure. There should only be one valid procedure in the specified file. Any other setup procedures and functions should be included in the file, which contains the PARALLEL IDL PROCEDURE. The finalization procedure is run after all of the parallel work procedures have been finished. Thus the finalization procedure can use the output of the parallel procedure as input if necessary. If the user desires to have no finalization procedure, enter "none" or "<none>" for this parameter. The user may use the "browse" button to specify the FINALIZATION PROCEDURE.

4. OUTPUT DIRECTORY - Enter the full path name of the directory which will contain the output data files for both the parallel and finalization procedures. Any files, which are input files for the finalization procedure must also be put in this directory, unless they will be generated by the parallel work procedures. If the specified directory does not exist at runtime, it will be created. If the directory does exist, files may be overwritten.

5. WORKER HOST ADDRESSES - Enter the hostnames of the nodes, which will do the parallel processing. Fully qualified hostnames may be either entered by typing in the appropriate box or by using the pull-down menu. The user may also supply the location of the IDL distribution by entering this information in the box labeled IDL PATH. The default location is displayed. Once the correct hostname and IDL location is specified, select the "ADD HOST" button to add a new worker node to the list. The user may add the same host twice. This is useful if the worker node has two processors, for instance. To remove a host from the list of workers, select the host to be removed in the table, and click the "REMOVE HOST" button. Recall that all worker hosts must be cross-mounted (as is true of the manager

host) and that all workers must have a working version of IDL, which can be licensed at runtime. The manager does not have to have IDL. The user may add as many worker hosts as is desired. The default configuration uses the manager hostname as the single worker host.

Clicking the "DONE" button will write the entered configuration to a save file and will reset the manager GUI with the new information. Note that the user may now click the "SAVE CONFIG" button in the SESSION CONFIGURATION panel of the manager GUI to name this configuration so that it may be reloaded at a future time. The new configuration is also written as the last configuration, and will be automatically loaded the next time the manager is launched.

Installation

1. Prerequisites for the manager
    - SSH. Since the manager communicates with the workers' hosts using SSH, you must have a login account on all hosts you plan to use in the farm.
    - Java. A Java virtual machine is required for the manager. Hosts acting as workers do not need Java. More info on Java and obtaining a JVM at http://java.sun.com.
    - Ant. The Ant build system is required to build the project from CVS. Information and downloads of Ant are available at http://ant.apache.org/.
2. Configuring the workers
    - Identify a file system that all workers cross mount.
    - Identify the install path of IDL for each worker. TaskManager has been tested with both IDL 5.6 and 6.0 workers

Running TaskManager from CVS

1. Login via "ssh -X" to a host with a file system that all workers cross mount.

Note: In order for workers to access the work tickets, they must share a common file system.

2. Get project from CVS with the command cvs -d :ext:volt.txcorp.com:/projects co taskdl
3. Change to the working directory:

cd taskdl

4. Run the Ant configuration tool:

ant

Note: if the 'ant' executable is not in your path, you may need to specify the pathname manually.

5. The TaskDL GUI opens up. The settings from the last session configuration executed are loaded into the fields. To run a session with the current settings press the Execute button and enter login information as prompted. To change the configuration settings, press the New Config button, and follow the instructions in the Setup Wizard.