Configuring Scientific Applications in a Heterogeneous Distributed System

Patrick T. Homer
Richard D. Schlichting

TR 93-29

ABSTRACT

Current scientific applications are often structured as a collection of individual software components that are manually executed on heterogeneous machines, with files being used to transfer data from one component to the next. Yet despite having the structure of a distributed application from the perspective of configuration management, the techniques and tools that have been used in this domain to address configuration have generally been minimal at best. Here, an approach to configuring scientific applications in a heterogeneous distributed system is described. The focus is on Schooner, an interconnection system that provides the programming model and base technology needed for realizing enhanced configurability. One key aspect of this technology is a machine- and language-independent interface specification that is used to generate interface code to bind components into the application and map them onto suitable host architectures. The other is a runtime system that implements support for both static and dynamic configuration. This paper describes the Schooner application model, outlines the method of creating component interfaces, and describes the runtime system and its various configuration options.

September 1, 1993

Department of Computer Science
The University of Arizona
Tucson, Arizona 85721

This work supported in part by the National Science Foundation under grant ASC-9204021. Homer is supported by the National Aeronautics and Space Administration under GSRP grant NGT-50966.

Configuring Scientific Applications in a Heterogeneous Distributed System

1. Introduction

A scientific application often involves multiple software components executing on multiple heterogeneous machines in a distributed environment. For example, one machine may be used to generate a computational grid, a second to execute the main computation using that grid, and a third to visualize the results. Yet despite having the structure of a distributed application from the perspective of configuration management, the techniques and tools that have been used in this domain to address configuration have generally been minimal at best. Most often, the user runs the components as individual programs, with files and manual file transfer being used to move the data from one component to another. Among the problems caused by this approach are an unreasonable burden placed on the user due to the potentially complex series of actions that must be carried out; poor performance due to the need to create the file, copy it to the next machine, and have it read by the next component; and difficulty in developing computational codes that can be used in multiple applications due to the need to handle a variety of file formats as input.

Schooner is an interconnection system designed to address these and other problems in the context of scientific applications [Homer92]. In Schooner, an application is viewed logically as a collection of independently developed components. Each component has a *code block* and an *interface*. The code block realizes the required application functionality and is implemented by the user; the interface realizes data conversion and other functionality required to execute components in a distributed heterogeneous environment and is generated automatically by the system. At execution time, the user selects the components that make up the application and specifies on which host each component is to be executed. The Schooner runtime then automatically creates connections among the components and manages the transfer of control and data needed for execution.

The most important aspects of Schooner from the point of view of supporting configurable applications— and hence the focus of this paper—are the code block interface and the runtime system. The key to solving the interface problem efficiently is the use of a machine- and language-independent type description language called UTS [Hayes89]. This language is used to write interface specifications, which are read by a stub compiler to generate the necessary interface functionality. The specifications are also used by the runtime system to ensure that interconnected components match in terms of the type of the values being transferred and in which direction.

Direct support for configuration is implemented in the runtime system, which has both static and dynamic configuration options. With static configuration, the user describes the configuration of the application—that is, the components and machines to be used—prior to execution. The runtime then starts the application and runs it to completion. With the dynamic option, the user has the additional ability to control the application during execution by adding, deleting, and moving components. This facility allows the user to exercise more interactive control, which can be especially useful for adapting the application to conditions that vary between different execution runs. Access to this dynamic configuration capability is provided through a collection of library routines or an interactive interface called the Controller.

This paper describes how Schooner can be used to configure scientific applications in a heterogeneous distributed system, as follows. First, Section 2 gives an overview of the Schooner model of scientific applications and the system itself. Section 3 then describes how the interface portion of a component is created and provides some details about the implementation of the interface. The Schooner runtime system and support for static configuration are the topic of Section 4, followed by details of the dynamic configuration options in Section 5. Finally, Section 6 covers related work, while Section 7 offers some conclusions.

2. A Model for Scientific Applications

2.1. Building Block Paradigm

A scientific application is especially well suited for being modeled as a collection of interconnected *components.* In this view, each component contains one computation code or data manipulation tool that accomplishes a specific task. The developer uses these components as building blocks, configuring an application from a palette of components and determining where each will execute. The runtime system then determines the communication mechanisms to be used and provides the ability to add, delete, and move components during execution. The communication mechanism logically connects the components and transparently converts data as needed for components on different architectures. In essence, this model characterizes a scientific application as a *heterogeneous distributed program* [Homer92].

Although this view has many advantages, perhaps the most important is that it supports the possibility of rich forms of interaction among the components. To exploit this potential, however, requires significant support for configuration. For example, consider an application involving a combination of a grid generation code, a fluid dynamics code, and a visualization tool. The process of using these components is often an iterative one. The grid generator, for instance, may be used multiple times to produce a range of grids in the search for the grid that produces the best results with the fluid dynamics code. The visualization tool would be invoked multiple times as well to view the results of the computation, often mapped over the grid, and perhaps to form a history of the results of a number of computation runs. A properly designed system would allow the user to build this application at execution time by selecting codes from a list of available versions of each type and then mapping each component onto the appropriate machine.

The NASA Numerical Propulsion System Simulation (NPSS) project [Claus91] is an example of a complex project where adequate support for configuration is imperative. The goal of the project is to provide better applications for use in designing jet engines. To accomplish this goal, the project is moving in two directions. One direction involves the development of improved codes for each functional part of an engine with a special emphasis on exploiting parallel hardware. The other direction involves the design of a *simulation executive*, a configuration system that allows the user to select the codes used to model each part of the engine, connect them into a whole engine, and map each code onto specific hardware. The simulation executive provides the ability to interact with each part of the engine to set needed parameters, as well as providing overall control of the simulation. The ability to configure an engine from available codes for the various parts allows the user to experiment freely with different codes, different parameters, and different engine operating constraints without the need to spend time creating connections between components for each run.

It is worth emphasizing again that this model of scientific applications is different from current common practice. Most often, each component is now executed separately with a data file used for data transfer, rather than having a direct data and control flow from component to component as in this model. This file-oriented paradigm solves some of the problems of configuring applications from components, but only at the expense of considerable (and repeated) effort on the part of the user. For example, files are easily transferred from host to host using network file systems or FTP services, which solves the problem of getting data from a producing component on one host to a consuming component on another. Similarly, data stored in files can be encoded as ASCII text, which solves the conversion problem inherent when heterogeneous architectures are involved. Yet to use files in this way, the user must perform all the configuration tasks, including host mapping and file transfer, and is responsible for ensuring data compatibility. In short, without configuration support, the user must make all the decisions at every step and, in effect, must implement the interface between the application's components.

2.2. The Schooner Interconnection System

Schooner is a component interconnection system that supports the construction of scientific applications according to the building block paradigm. An application is configured by the user from a collection of independently developed components, with various options provided for binding components into applications and mapping them onto the available hardware. Schooner handles the connections between components, including providing a name service to locate the component, data conversion, and selection of the best network protocol.

To allow true configurability in an Internet-wide environment, an interconnection system must support heterogeneity in programming models, programming languages, and host and network architectures. In addition, the focus on scientific applications imposes two additional requirements. First, the system must be easy to use, so that the computational scientist who actually constructs the application will be willing to make the transition from more traditional methods. Second, the impact on the source code for each component in the system must be minimized. These codes are often independently developed and the source may be unfamiliar to the programmer, maintained by a different group, or simply unavailable.

Schooner meets these requirements—configuration, heterogeneity, ease of use, minimum impact on source—in the way it supports the building block paradigm. As noted in the Introduction, each component is composed of two parts: a *code block* and an *interface.* Each block is a code or tool the scientific programmer wants to use in the application. In the fluid dynamics application described earlier, for example, the grid generator, fluid dynamics computation, and visualization tool would each be one code block. The interface is a stub and collection of library routines that is attached to the block to carry out data conversions and implement the message passing required to transfer data between the processes that execute the components. Each interface is described by an *interface specification* that defines the services made available to other components and the services the component requires that are elsewhere in the application. At runtime, the Schooner system uses these specifications to match requirements and form the necessary connections between the component implementing a given service and its users (Figure 1).

As an example of the use of interface specifications, consider a grid generation component that produces grids in multiple formats to satisfy the needs of different computation and visualization tools. The specification would have a description for each grid type. Similarly, the specification for a fluid dynamics component will have a description of its input grid. When an application is then created from the grid and fluid dynamics components, the Schooner runtime performs type checking to ensure the compatibility of the output and input grids. Doing such a check with an interface specification is difficult at best.

Schooner realizes control flow in the application using the remote procedure call (RPC) paradigm. Thus, the interface specification is simply a description of the procedures made available by a component for external invocation, as well as those external procedures that the component invokes. The runtime system configures applications by starting components on their target machines, collecting information from the interface specification of each component to build a location database, and checking the appropriate procedure specifications for type compatibility. As will be seen below in Section 5, the runtime also provides the ability to add, delete, and move components.

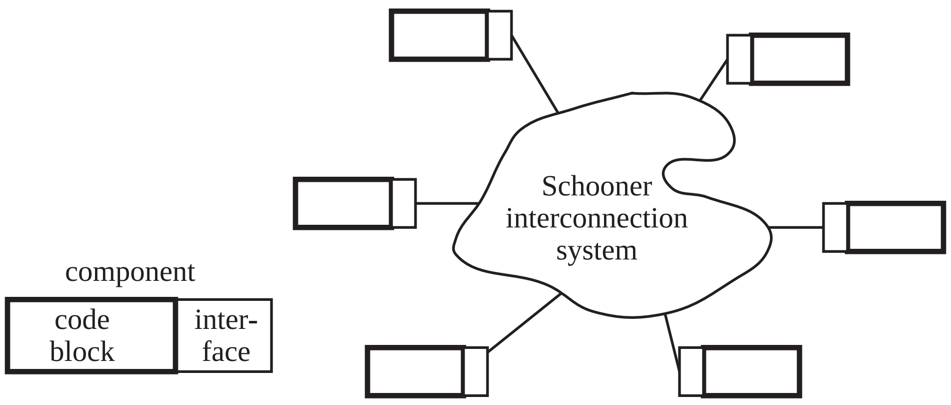

Figure 1 — Creating an application from components

3. Interface Construction

The interface code for a component and its associated specification are an important aspect of the infrastructure needed by Schooner to support configurability. To realize the required functionality, the system provides two services: a simple type description language and a collection of stub compilers and libraries. The type language is used to write the interface specification, while the stub compiler reads this specification and generates the interface code itself. Linking the interface with the code block and the Schooner libraries produces an executable component that can be configured into applications.

3.1. Interface Specification

The Universal Type System (UTS) [Hayes89] provides a specification language for defining component interfaces and a library for converting data formats across machine boundaries. The type language defines interfaces using a syntax that is independent of both the execution host architecture and the programming language used in the code block. Among other things, this feature allows the same interface to be used for different versions of a code block, such as, for example, a sequential and parallel implementation.

The interface specification describes those procedures the code block makes available for other components to call (*export* procedures) and those remote procedures needed by the code block (*import* procedures). UTS uses a Pascal-like syntax to describe the parameters for the two types of procedure. For example, the interface specification for the component that exports a fan code in the previously-mentioned NPSS jet engine simulation is as follows:

```
export fann prog(
   "pin"   val float,   "pout"   val float,   "tin"    val float,
   "tout"  val float,   "xspool" val float,   "cshift" res float,
   "hout"  res float,   "eout"   res float,
   "pmap1" var array[14] of float,
   "pmap2" var array[14, 12 3] of float,
   "pmap3" var array[3] of float)
```

The keywords `val`, `var`, and `res` indicate the direction of data transfer for each parameter, while `export` indicates that the component will provide this procedure for other components to call. Components wishing to call the procedure `fann` would then have a matching specification that uses the key word `import` rather than `export.` The words in quotes are comments, used here to denote the parameter names.

The UTS language allows specification of all the typical simple data types (integer, float, double, character, boolean), as well as the structured types array, record, and string. Procedure values are also supported. Schooner treats each such value as a capability to the named procedure, thereby allowing other components to perform a call back operation by invoking the capability. Data types that are not present explicitly in UTS can often be described; for example, the FORTRAN complex type can be described as:

FORTRAN	UTS
complex	record{float, float}
double complex	record{double, double}

In most cases, the only difference between the import and export specifications is the use of the keyword `import` or `export.` UTS does, however, allow the import to be, in essence, a subset of the export. This can increase the usefulness of an exported procedure for configurable applications by allowing it to handle a variety of inputs, rather than the alternative of writing a collection of procedures all performing nearly the same operation on slightly different inputs. As a small example, a procedure with the specification

```
    export example prog(val (integer or float))
```

can accept either an integer or a float as its argument. A component wishing to call this procedure can use either of the following specifications:

```
    import example prog(val integer)
    import example prog(val float)
```

3.2. Creating the Interface

As mentioned, a stub compiler is used to generate the code for the component interface. There is one stub compiler per supported language, currently C and FORTRAN.[1] The compiler reads the specification file and produces a machine-independent stub that sits between the code block and the Schooner libraries (Figure 2). The stub includes calls to two libraries, the UTS library and the Schooner communications library. The UTS library implements data conversions, while the communications library marshalls the arguments into messages, locates the remote procedures, and sends and receives the calls and replies. The stub and the code block are then compiled and linked with the UTS and communications libraries. If there are to be multiple instances of a code block, as would be the case if it is to run on more than one architecture or when more than one algorithm is available, the user compiles and links the appropriate code block with the stub on each target machine. The differences in storage formats and communication requirements are handled transparently by the two libraries.

The data conversion routines in the UTS library convert between a standard format based on the IEEE specification and each supported architecture's storage format. These conversion routines are automatically supplied in most cases by the stub compiler, using the description given in the specification file. The resulting stub contains the appropriate calls to the library for each parameter in turn and correctly handles function return values and val and res parameters that travel in only one direction.

For situations where automatic conversion is not possible, the programmer can access the UTS library directly. Such a situation would arise, for example, when transferring a dynamically allocated array where the size of the array is not known at compile time. It also occurs when a C

[1]The predecessor MLP system [Hayes88] also supported Pascal, Icon [Griswold90], and Emerald [Black87].

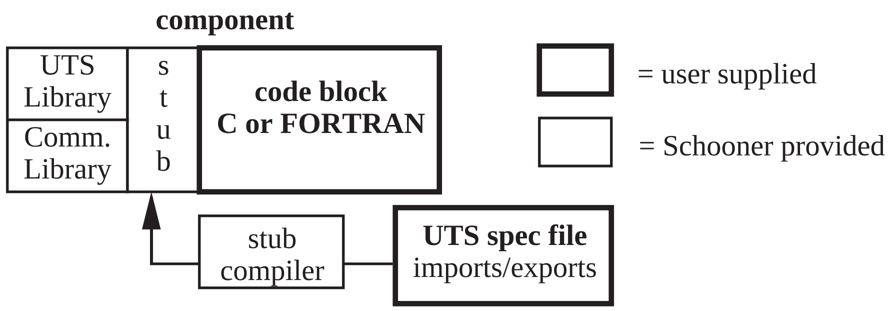

Figure 2 — Component Interface

struct (described in UTS as a record) is passed as data to a FORTRAN component. While the stub for the C component handles the conversion automatically in this case, the stub in the FORTRAN component passes the encoded UTS record directly to the code block. The programmer then explicitly invokes the UTS library to decode and extract the individual fields in the record as needed.

The UTS data representation itself uses a tagged format, where each data value in the encoded representation is marked by a tag indicating its type. Since this feature means that the type of a data value can be determined without *a priori* knowledge, each received data item can easily be type checked against the import specification, adding an extra safety margin important in systems where components may be reconfigured dynamically. The above example of a procedure accepting multiple types is another case where the use of tagged data is necessary. The tag allows the component to determine which data type from those listed is actually received. The component can then call the correct conversion routine to extract the parameter's value from the message. The tags have also proven very useful for debugging applications, since the contents of messages can be read and interpreted at any point.

The Schooner communications library implements the interface to Schooner's dynamic name resolution and interprocess communication facilities, both of which are important for supporting configurable applications. The interface to the name resolution mechanism consists of two routines, one that registers a component with the Manager process (described in the next section) and the other that queries the Manager when an imported procedure is invoked for the first time. The interface to the interprocess communication facilities consists of a send operation for outgoing calls and a receive operation for incoming calls. The send operation takes the marshalled arguments provided by the stub, places them in the message along with return address information, sends the message to the remote component, and handles the return message, passing the marshalled return values to the stub as needed. The receive operation is structured analogously.

4. Static Configuration

Schooner's static configuration option is designed to meet the minimum requirements of allowing the user to defer until execution time the two choices of what components to bind into the application and how to map the components onto the available hosts. Before describing the details of how this is done, however, we first describe the Schooner runtime system, which supports both static and dynamic configuration. The runtime system allows the user the flexibility to specify the components and hosts to be used for the computation, while hiding the details of startup and communication.

4.1. Schooner Runtime System

In addition to the processes that actually execute components, the Schooner runtime employs two other types of processes. The first is the Manager mentioned above; there is one such process per application and it typically executes on the user's home machine. The second is a Server process; there is one such process per host machine used in the application.

The Manager is the central coordinator of the application, handling the tasks of mapping components to specified hosts and binding components into an application. The mapping task is carried out with the cooperation of the Server. Specifically, for each component, the Manager sends a request to the Server on the target host, passing the name and path of the executable. The Server then starts the component with the location of the Manager as an argument. Any startup errors are reported back to the Manager.

Components become bound into the application by the registration procedure mentioned above. That is, upon startup, each component sends a message to the Manager containing the UTS specifications for all procedures that it exports. This information is then entered in a procedure name database, which is used to respond to subsequent name resolution requests. Once all components have registered with the Manager, the binding of the components into an application has been completed.

4.2. Command Line Execution

The format for static configuration is modeled after the normal method of executing an application from the command line. In particular, the user lists the {component, host} pairs as arguments when invoking the Manager as a command. Schooner also provides for passing command line arguments to the main procedure of the application.

The Manager starts the components and, when all the registration calls have arrived, begins execution by an invocation to the component exporting the main procedure. Control is then passed from component to component through the chain of remote calls as execution proceeds. As described above, name resolution requests are forwarded to the Manager whenever an imported procedure is invoked for the first time. In addition to the name of the procedure, this request includes the appropriate UTS import specification. Upon receiving this request, the Manager looks up the procedure name and performs type checking by comparing the import specification in the request with the export specification in the database. If the match is legitimate, the location of the target procedure is returned to the calling component, which proceeds with the remote invocation. This location information is cached at the calling component so that future calls proceed directly.

When execution terminates, normally through the main procedure returning, the Manager terminates each of the components in the application. Shutdown will also occur in the event of an error. In addition to the usual errors that occur in an application, e.g., floating point exceptions, Schooner will also report startup and type checking errors.

5. Dynamic Configuration

Schooner also supports dynamic configuration, that is, the ability to modify the binding among components and their mapping to hardware after an application begins execution. In particular, components can start independently of the Manager and join an application, can be started during execution from another component, and can be moved from one host to another. Thus, in this option, the mapping of component to host is not permanent, while the binding of components into an application becomes open-ended and more flexible.

Two methods are available in Schooner for controlling applications in a dynamic mode. One is a set of library routines that allow the user access to the Manager's configuration functions. The second is a Controller process that provides an interactive interface for creating, executing, and tracing applications. The implementation of the library routines has been completed, while the Controller is an ongoing project.

5.1. Library Support

Schooner provides a set of library routines that supports the configuration requests mentioned above: component joining, startup requests, and transfer from one host to another. Joining a Schooner application is accomplished through a call to a registration routine. This call allows a component to establish communications with the Manager and informs the Manager of the exported procedures available from the component. Once added to the application, the component's procedures can be called from other components. As will be shown below, this method is particularly useful when working with applications that have to start on their own, rather than be started by the Manager. Essentially, the library call allows a Schooner identity to be added to a component that has other identities as well.

A start component library routine allows an application to request the start of additional components. The invocation specifies both the executable to be used and the host. The new component will be bound into the application by the Manager and its procedures made available. Once the start component call has returned, any of the components in the application are free to call the procedures exported by the new component. This allows the application to determine its configuration based on factors known after execution has begun.

Schooner has a limited ability to move a component during execution from its current host to a new host. The move ability is limited by the philosophy of minimizing the changes needed in the user's source code, since without participation from the code block and access to the internal call stack, general process migration is impossible [Hofmeister93, Douglis91]. However, Schooner's use of RPC allows a component to move when it is between calls. To maintain the correct semantics, the procedures in the component need to be stateless; that is, the component cannot contain variables that retain values across calls.[2] Any component in the application can issue the move command. Upon receipt, the Manager will first shut down the current version of the component by sending it a quit message and removing the names of its procedures from the database. The Manager then starts the component via a call to the Server on the target host. Once the new version of the component has performed a registration call, the Manager updates its database and replies to the move command. The application then proceeds with its execution. The local procedure name caches in each component are not automatically updated when a component is moved. Instead, an attempt to call a procedure in a moved component will fail and result in another mapping request to the Manager to inquire about an updated location. The new location is then cached locally and the call proceeds.

An example of the use of this library interface to support configurability is the prototype simulation executive constructed for the NPSS project using Schooner and the AVS scientific visualization system [AVS92, Homer93, Homer92]. AVS employs a Network Editor that allows the user to create programs from a palette of modules. The programs are formed by dragging modules into the workspace and connecting them with lines marking the flow of data through the program. When executed, AVS invokes each module as indicated by the data flow and the module is then able to extract information from the data flow, process it, and pass data on to the downstream modules. AVS also makes use of *widgets* that allow the user to directly input values into a module. Widgets take on a variety of appearances in the interface, including dials, sliders, type-in boxes, and radio selection buttons. The value of a widget can be changed by the user, in which case it is read as a parameter when the module is next executed. In addition to the supplied modules, AVS allows a user to write modules that can make use of the widget interface and data flow network.

The prototype NPSS simulation executive uses AVS and its data flow network as an execution framework and Schooner for connecting to computational codes on remote, possibly heterogeneous, machines. Figure 3 shows an AVS network that implements a one-dimensional model of a jet engine. Here, the actual computation associated with the nozzle module is done on a remote host, with the choice of machine specified using the widgets on the left.[3] The remote execution is transparent to AVS, while from a Schooner perspective, the nozzle module and the remote computation are separate components that are bound into the same application. Of course, the remote call in this case originates from the AVS module and is received and executed by the component on the remote machine.

The configuration functionality in the prototype simulation executive is implemented by invoking the Schooner library routines within the AVS module. After the module is initiated by AVS, it invokes the registration routine to identify itself to the Manager as a Schooner

[2]A future addition planned for Schooner will support state variables that are globally defined within the component. A UTS description of these variables would be used to transfer their values to the new component during a move.

[3]Of note is that the collection of machines is widely distributed geographically, with some at NASA Lewis Research Center in Ohio and others at The University of Arizona.

Figure 3 — Nozzle Widgets and AVS network

component. The module then reads the path name of the executable to be started as the computation component and the name of the remote host from the appropriate widgets (see Figure 3). These values are used as arguments in an invocation to the Manager requesting that the remote component be initiated and bound into the application. As execution proceeds, the nozzle module makes calls as needed to the procedures exported by the remote component, with the Schooner runtime system implementing the transfer of control and data.

5.2. Controller

The Controller is an X-windows interface to the Manager that provides interactive access to the Manager's configuration library, and hence, the ability to configure the application dynamically. It satisfies this role by providing four features:

- Mapping of components onto hosts.
- Binding of components into applications.
- Execution of the application or individual components
- Tracing of the remote procedure calls and replies during execution

Of these, the first two are implemented, while the last two are in progress.

The user creates an application with the Controller by starting the desired components and binding them together using its visually-oriented interface. The user starts a component by specifying the executable and the host, either in a type-in window or by selecting the appropriate name from a menu. To implement this functionality, the Controller invokes the Manager's start component function and reports any error messages from the Manager, such as a bad host name or file not found, allowing the user to try again.

The Controller also allows the user to bind components into an application as they are started. This includes components such as AVS modules that are started independently. In this case, the Controller receives the relevant information from the Manager when the component registers and displays it to the user in a suitable format. Binding such a component into the application makes its procedures available for invocation from other components and vice versa.

Application execution can be started from the Controller as well. When the user makes such a request, the Controller sends a message to the Manager asking that it invoke the main procedure. The user is also able to pass command line arguments to the main procedure through the Controller. Components can subsequently be added to the application by the user as execution proceeds. For example, a name resolution request made to the Manager looking for a procedure that is not in the database will prompt a warning message to the Controller. At this point, the user can start a component containing the needed procedure, bind it into the application, and then allow execution to proceed.

In addition to calling the main procedure, the Controller is able to call any exported procedure, allowing the testing of parts of a computation without the need to have all the pieces working. The UTS export specifications for the procedures allow the Controller to prompt the user for values to assign to each of the parameters. The Controller then calls the procedure and receive the reply, again using the UTS specification to display the results for the user.

A final Controller feature is the ability to trace the remote procedure calls made within an application. To do this, each component forwards copies of its calls and replies to the Controller, which displays them to the user. The UTS specification and tagged presentation are again useful in this context since they facilitate the decoding and display of the arguments and return values found in these messages.

6. Related Work

A number of other projects are investigating issues of configuration and heterogeneity in scientific applications. The Polygen system [Callahan91] supports configuration through the use of specification files. It uses a specification file for each module similar to the UTS specification files in Schooner. To configure an application, Polygen also uses a specification file, called a composite specification, that describes the collection of modules and the bindings among them to create the application. The Polylith software bus [Purtilo90] is then used to create the application and execute it, following the directions in the specification files. The software bus encapsulates the heterogeneity and distribution aspects of the application, handling the establishment of communications and creation of the modules. Polygen supports automatic compilation of modules, including rules for recognizing when multiple modules can be combined into the same executable based on information provided in the module specifications.

Schooner differs from Polygen in handling most of the configuration at runtime rather than through specification files for the application. The Schooner runtime uses a simpler system, not dealing with compilation issues or trying to package multiple modules into one executable. This is in keeping with our philosophy of providing a tool for a scientific programmer to use— that is, someone familiar with compilation techniques—and serving primarily as an interconnection tool.

Systems such as PVM [Sunderam90], p4 [Butler92], and APPL [Quealy93] support distributed and parallel processing involving multiple processes. They seek to implement parallel algorithms using a message passing paradigm on parallel machines and/or across networks of computers. PVM and p4 also provide support for heterogeneity through conversions of data types across machine boundaries. HeNCE [Beguelin91] is a tool built on top of PVM using a visual programming interface to construct programs. The interface allows the user to indicate various parallel constructs, such as the ability to fan a set of jobs out across processes and collect results. It also allows the user to assign tasks to machines. It provides a post-run trace and playback mode for debugging applications.

Schooner differs in orientation from these systems, seeking to ease the tasks of adapting components and configuring them into applications rather than exploiting parallel environments for improving execution times. Indeed, a program using PVM, for example, can become a component within a Schooner application. The combination of RPC and the UTS type language simplifies the construction of the interface by providing a familiar procedural format and supporting the automatic conversion of most data values.

Several other research projects [Chen93, Freund93, Khokhar93, Wang92] are also studying the issue of heterogeneity in scientific applications. In particular, they are investigating ways to recognize heterogeneity in an application and automatically partition the application to execute on a (virtual or real) heterogeneous machine. This work involves both the exploration of hardware techniques to incorporate multiple types of processors within the same machine, and software techniques to recognize heterogeneity at a low-level and automatically select the right processor for each portion of the overall algorithm. Our work differs in the direction we go to find heterogeneity. As described in Section 2, rather than looking within a program, our model is to consider the programs themselves to be the building blocks of a larger application. As a result, Schooner’s support for heterogeneity is used at a higher level for assembling applications from components.

7. Conclusions

Schooner is an evolving system that supports the configuration of scientific applications in a heterogeneous distributed system in a variety of ways and at a variety of levels. At a technical level, the system includes provisions for configuring applications from components either statically or dynamically. The static option is a command line interface in which the user specifies the components and target machines; the program then executes to completion, with Schooner handling the inter-host communication and heterogeneity aspects transparently to the user. The dynamic options are based on a collection of library routines that interact with the Manager to perform configuration activities such as starting, registering, and terminating components. These routines can either be accessed directly, as was done when AVS and Schooner were used in tandem to create the prototype NPSS simulation executive, or through a Controller process that provides an easy-to-use visual interface. All these configuration features are based on and facilitated by underlying features of Schooner, including the structure of both the interface code and the runtime system.

Perhaps more important, however, is the support that Schooner provides for configurable applications at the conceptual level. The Schooner application model, in which computational codes are viewed as building blocks in a larger application, encourages the end user to think of configuration as an integral part of the development process. Effecting a change in mindset along these lines is one of the biggest remaining challenges, especially in scientific computing where the focus is primarily on the results of the application rather than the process itself.

Acknowledgements

This work was performed in part on computing resources at the Advanced Computing Concepts Laboratory and the Computer Services Division at NASA Lewis Research Center.

References

[AVS92] Advanced Visual Systems Inc. *AVS Developer's Guide* (Release 4.0), Part number: 320-0013-02, Rev B, Advanced Visual Systems Inc., Waltham, Mass., May 1992.

[Beguelin91] Beguelin, A., Dongarra, J. J., Geist, G. A., Manchek, R., and Sunderam, V. S. Graphical development tools for network-based concurrent supercomputing. *Proc. Supercomputing '91*, Albuquerque, NM (Nov. 1991), 435-444.

[Black87] Black, A., Hutchinson, N., Jul, E., Levy, H. and Carter, L. Distribution and abstract types in Emerald. *IEEE Trans. on Softw. Eng. SE-13*, 1 (Jan. 87), 65-76.

[Butler92] Butler, R. and Lusk, E. User's guide to the p4 parallel programming system. ANL-92/17, Argonne National Laboratory, Argonne, IL, Oct. 1992.

[Callahan91] Callahan, J. R., and Purtilo, J. M. A packaging system for heterogeneous execution environments. *IEEE Trans. on Softw. Eng. SE-13*, 6 (Jun. 1991), 626-635.

[Chen93] Chen, S., Eshaghian, M., Khokhar, A., and Shaaban, M. A selection theory and methodology for heterogeneous supercomputing. *Proc. Workshop on Heterogeneous Processing*, Newport Beach, CA (Apr. 1993), 15-22.

[Claus91] Claus, R.W., Evans, A.L., Lylte, J.K., and Nichols, L.D. Numerical Propulsion System Simulation. *Computing Systems in Engineering 2*, 4 (Apr. 1991), 357-364.

[Douglis91] Douglis, F., and Ousterhout, J. Transparent process migration: Design alternatives and the Sprite implementation. *Software—Practice and Experience 21*, 8 (Aug. 1991), 757-785.

[Freund93] Freund, R. F., and Siegel, H. J. Heterogeneous processing. *Computer 26*, 6 (June 1993), 13-17.

[Griswold90] Griswold, R. and Griswold, M. *The Icon Programming Language*, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[Hayes88] Hayes, R., Manweiler, S., and Schlichting, R.D. A simple system for constructing distributed, mixed-language programs. *Software—Practice and Experience 18*, 7 (July 1988), 641-660.

[Hayes89] Hayes, R. *UTS: A Type System for Facilitating Data Communication*. Ph.D. Dissertation, Dept. of Computer Science, Univ. of Arizona, Aug. 1989.

[Hofmeister93] Hofmeister, C., White, E., and Purtilo, J. Surgeon: A packager for dynamically reconfigurable distributed applications. *IEE Software Engineering Journal 8*, 2 (Mar. 1993), 95-101.

[Homer92] Homer, P.T., and Schlichting, R.D. A software platform for constructing scientific applications from heterogeneous resources. Tech. Report 92-30, Dept. of Computer Science, Univ. of Arizona, Nov. 1992.

[Homer93] Homer, P. T., and Schlichting, R. D. Supporting heterogeneity and distribution in the Numerical Propulsion System Simulation project. *Proc. 2nd Intl. Symp. on High Performance Distributed Computing*, Spokane, WA (Jul. 1993), 187-195.

[Khokhar93] Khokhar, A. A., Prasanna, V. K., Shaaban, M. E., and Wang, C. Heterogeneous computing: Challenges and opportunities. *Computer 26*, 6 (Jun. 1993), 18-27.

[Purtilo90] Purtilo, J. M. The Polylith software bus. Institute for Advanced Computer Studies and Dept. of Computer Science, Univ. of Maryland, UMIACS-TR-90-65, May 1990.

[Quealy93] Quealy, A., Cole, G. L., and Blech, R. A. Portable programming on parallel/networked computers using the Application Portable Parallel Library (APPL). NASA Technical Memorandum 106238, Jul. 1993.

[Sunderam90] Sunderam, V. S. PVM: A framework for parallel distributed computing. *Concurrency—Practice and Experience 2*, 4 (Dec. 1990), 315-339.

[Wang92] Wang, M., Kim, S., Nichols, M., Freund, R., Seigel, H., and Nation, W. Augmenting the optimal selection theory for superconcurrency. *Proc. Workshop on Heterogeneous Processing*, Beverly Hills, CA (Mar. 1992), 13-22.