2001

DCAPS - Architecture for Distributed Computer Aided Prototyping System

Berzins, V.; Luqi; Ge, J.; Shing, M.; Auguston, M.; Bryant, B.; Kin, B.

IEEE

DCAPS – Architecture for Distributed Computer Aided Prototyping System[1]

Luqi, V. Berzins, J. Ge, M. Shing, M. Auguston[2], B. Bryant[3], B. Kin
Department of Computer Science
Naval Postgraduate School
833 Dyer Road
Monterey, CA 93943 USA
{luqi,berzins,gejun,mantak,auguston,bryant,bkkin}@cs.nps.navy.mil

Abstract

This paper describes the architecture for the distributed CAPS system (DCAPS). The system accomplishes distributed software prototyping with legacy module reuse. Prototype System Description Language (PSDL), the prototyping language, is used to describe real-time software in the DCAPS system. PSDL specifies not only real-time constraints, but also the connection and interaction among software components. Automatic generation of software wrappers and glue is applied for the normalization of data transfer between legacy systems. Implementation of the DCAPS communication layer is based on the JavaSpaces™ library. DCAPS supports collaborative prototype design in a distributed environment.

1 Introduction and objectives

The value of computer-aided prototyping in software development is clearly recognized. It is a very effective way to gain understanding of the requirements, reduce the complexity of the problem and provide an early validation of the system design. Bernstein estimated that for every dollar invested in prototyping, one could expect a $1.40 return within the life cycle of the system development [1]. To be effective, prototypes must be constructed and modified rapidly, accurately, and cheaply. Computer aid for rapidly and inexpensively constructing and modifying prototypes makes it feasible [2].

With advances in wide area networks, there is a need for methods and tools to produce distributed, heterogeneous, and network-based systems that are reliable, flexible and cost effective. Many of these systems are COTS based (commercial off-the-shelf, including "legacy systems"), consisting of a set of subsystems, running on different platforms that work together via multiple communication links and protocols [3][4]. The use of COTS components shifts problems from software development to software integration and interoperability. Builders of COTS-based systems often have no control over the network on which components communicate. They have to work with available infrastructure and need tools and methods to assist them in making correct design decisions to integrate COTS components into a distributed network based system.

Furthermore, as software development has evolved into national and even global cooperative efforts with the explosion of the Internet and World Wide Web, the need for an effective distributed development environment to support such geographically dispersed enterprises became critical. The support is needed both for the distributed design and demonstration of real time system prototypes.

This paper addresses distributed rapid prototyping support for heterogeneous and network-based systems. It presents the underlying architecture to support the specification and automatic generation of codes to integrate and execute COTS components across a heterogeneous network.

2 Motivation and related work

2.1 Prototyping

The demand for large, high quality systems has increased to the point where a quantum change in software technology is needed [5]. Requirements and specification errors are a major cause of faults in complex systems. Rapid prototyping is one of the most promising solutions to this problem. Completely automated generation of prototypes from a very high-level language is feasible and generation of skeleton programming structures is currently

[1] This research was supported in part by the U. S. Army Research Office under contract/grant numbers 35037-MA and 40473-MA.

[2] On leave from Computer Science Department, New Mexico State University, USA

[3] On leave from Department of Computer and Information Sciences, University of Alabama at Birmingham, USA

common in the computer world. One major advantage of the automatic generation of codes is that it frees the developers from the implementation details by executing specifications via reusable components [5].

An integrated software development environment, named Computer Aided Prototyping System (CAPS) [6] has been developed at the Naval Postgraduate School for rapid prototyping of hard real-time embedded software systems, such as missile guidance systems, space shuttle avionics systems, software controllers for a variety of consumer appliances and military Command, Control, Communication and Intelligence (C3I) systems [7]. Rapidly constructed prototypes are used to help both the developers and their customers visualize the proposed system and assess its properties in an iterative process. The heart of CAPS is the Prototyping System Description Language (PSDL). It serves as an executable prototyping language at the software architecture level and has special features for real-time system design. Building on the success of the Computer Aided Prototyping System (CAPS), the DCAPS model also uses PSDL for specification of distributed systems and automates the generation of interface codes with the objective of making the network transparent from the developer's point of view.

2.2 PSDL and CAPS

PSDL, a prototype description language [8], to describe the real-time software has an open structure so that the user is able to define new properties for software components, such as newly added network configurations. PSDL allows the specification of both input and output guards to provide conditional execution of an operator and conditional output of data. Guards can include conditions on timers that measure duration of system states, and can allow operators to execute only when fresh data has been written to an input stream. Real-time applications, design flexibility, and code reuse motivate the timing and non-procedural control constraints of PSDL. Each time critical operator has a *maximum execution time* constraint, representing the maximum time the operator may need to complete execution after it is fired, given access to all required resources. In addition, each periodic operator has a *period* and a *deadline*. The period is the interval between triggering times for the operator and the deadline is the maximum duration from the triggering of the operator to the completion of its operation. Each sporadic operator has a *maximum response time* and a *minimum calling period*. The minimum calling period is the smallest interval allowed between two successive triggering of a sporadic operator. The maximum response time is the maximum duration allowed from the triggering of the sporadic operator to the completion of its operation. To model distributed systems, PSDL also provides the option of specifying the *maximum delay* associated with any data stream.

CAPS prototypes a software system in the following steps. First, the user selects software components from the reusable component libraries to construct the prototype in a graphic editor. This prototype is saved as a plain text file in PSDL format. The user may also use the graphical user interface (GUI) generator provided by CAPS to create a new GUI for demonstrating and observing the behavior of the prototype. Then, the translator and scheduler work on this PSDL file to generate the wrapper/glue code [9] and dynamic/static schedules [10] respectively. Both the source code of reusable components and automatically generated source code will be compiled together to get the executable. It will be run in the DCAPS environment in order to check both execution correctness and the real-time requirements. As described above, CAPS consists of various prototyping tools to provide all these functionalities. They play different roles during the prototyping process. For example, the scheduler just needs the timing constraints and execution order for every component, while the translator does not care about information other than the network configurations and data type definitions.

In order to automate the integration of COTS in a distributed environment, we need to enhance the modeling capability of PSDL to describe the special operating requirements of the COTS components and the quality-of-service characteristics for the target networks. The enhancement is done via the open syntax provided by the vertex property and edge property of the PSDL graph. Figure 1 shows an example where *the monitor_ environment* and the *temperature_control* operators are realized by COTS components that must run on a Windows NT™ operating system and the *valve_control* operator is realized by a COTS component that must run on a SunOS™ operating system. Furthermore, the *valve_adjustment* data must be transmitted via network links with high security and low latency while the *temperature* data can be transmitted via network links with low security and higher latency. When new properties are introduced in the PSDL descriptions of a prototype, for instance to prototype networked software, some tools must be updated while the rest stay the same. Therefore, the architecture of CAPS must consider the evolution of its own components.

CAPS tools were originally developed in the SunOS operating system for components located on one processor. To avoid the complexity of migrating the whole system to a new operating system, CAPS now has to work in a distributed and heterogeneous environment

2.3 Transaction handling in distributed systems

Building a networked application is entirely different from building a stand-alone system in the sense that many additional issues need to be addressed for smooth functioning of a networked application. Networked systems are

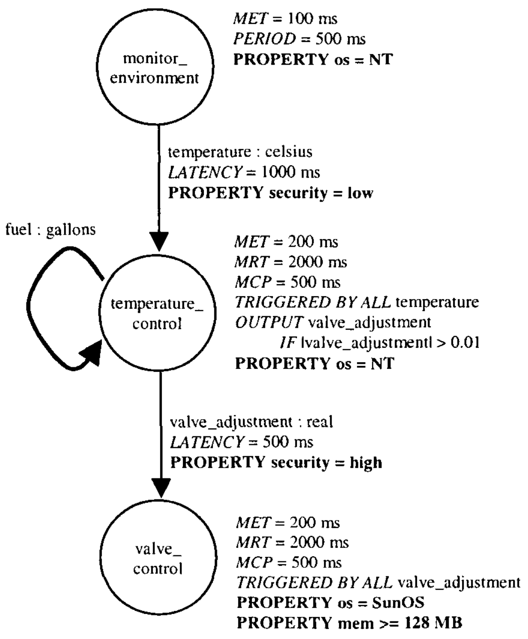

Figure 1. PSDL specification with additional properties

also susceptible to partial failures of computation, which can leave the system in an inconsistent state.

Proper transaction handling is essential to control and maintain concurrency and consistency within the system. Yang [11] has examined the limitation of hard-wiring concurrency control into either the client or the server. He found that the scalability and flexibility of these configurations is greatly limited. Hence, he presented a middleware approach: an external transaction server, which carries out the concurrency control policies in the process of obtaining the data. Advantages of this approach are: 1) The transaction server can be easily tailored to apply the desired concurrency control policies of specific client applications. 2) The approach does not require any changes to the servers or clients in order to support the standard transaction model. 3) Coordination among the clients that share data but have different concurrency control policies is possible if all of the clients use the same transaction server. PSDL already has a very simple and effective transaction model [12][13] . Transactions are determined by the simple rule that the effect of firing a composite operator must always be equivalent to executing it as a simple atomic action. Optimizations may introduce concurrency and interleave substeps only if that can be done consistently with this rule.

The DCAPS implementation architecture uses the same approach, by using an external transaction manager such as the one provided by SUN in the Jini™ [14] model. All transactions used by the clients and servers are created and overseen by the manager.

2.4 JavaSpaces model

JavaSpaces [14] is a mechanism based upon the Tuple Space model [15] to support coordination among a loosely coupled collection of distributed software systems. Tuples are typed data structures. Collections of tuples exist in a shared repository called a tuple space. Communication takes place in a tuple space shared among several processes; each process can access the tuple space by inserting, reading or withdrawing tuples.

When taking or reading objects, processes use a simple value-matching lookup to find the objects that matter to them. If a matching object is not found immediately, then a process can wait until one arrives. Unlike conventional object stores, processes do not modify objects in the space or invoke their methods directly. To modify an object, a process must explicitly remove it, update it, and reinsert it into the space. During the period of updating, other processes requesting for the object will wait until the process writes the object back to the space. This protocol for modification ensures synchronization, as there can be no way for more than one process to modify an object at the same time. However, it is possible for many processes to read the same object at the same time.

The main benefits of JavaSpaces from the point of view of DCAPS are:

- Spaces are persistent: Spaces provide reliable storage for objects. Once stored in the space, an object will remain there until a process explicitly removes it. This allows a system to perform communication with other systems which may not have begun running yet.
- Spaces are transactionally secure: The JavaSpaces technology provides a transaction model that ensures that an operation on a space is atomic. Transactions are supported for single operations on a single space, as well as multiple operations over one or more spaces.
- Spaces allow exchange of executable content: While in the space, objects are just passive data, however, when we read or take an object from a space, a local copy of the object is created. Like any other local object, we can modify its public fields as well as invoke its methods.

- Spaces transcend network topologies: Not only do senders and receivers of messages not need to know each others identities, they also may be located anywhere on the network as long as both have access to the common space.
- Spaces support for time-outs for data.

These properties greatly facilitate the communication layer to be inserted by DCAPS between the various legacy systems being integrated, and ensure the interoperability of these systems.

3 Architecture

3.1 Design time slice of the architecture

The design phase in the DCAPS environment emphasizes the retrieval of PSDL specifications, legacy code (when needed) and distributed resource configuration descriptions both from the server's Project repository and client side directories (Figure 2). DCAPS allow users to model, develop, execute and evaluate prototypes of the proposed systems from different hardware platforms with different operating environments via a web interface shown in Figure 3, where the hyperlinks on the left side of the web-page allow visitors to access information about CAPS, PSDL and request accounts, while the hyperlinks on the right are password protected and can only be accessed by authorized users.

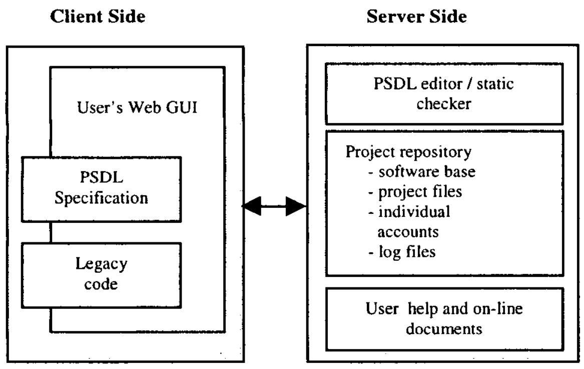

Figure 2. Design Time Slice of Architecture

The Java^{TM}-based user GUI ensures that the basic design time tools, such as the graphical PSDL editor, static checker, user help and on-line documentation, and demos are available for clients to run on heterogeneous platforms. An integral part of the Project repository is also individual account information and log files from previous prototyping sessions.

3.2 Compile time slice of the architecture

In the compilation phase of DCAPS, the client-side legacy system and PSDL specification of that system, its interface to the external environment, and the distributed resource configuration under which that system is to be run, will be input to the compiling tools residing on the server side (Figure 4). In actuality, these compilation tools may be downloaded to the client side, e.g. using a Java applet, to achieve the compilation.

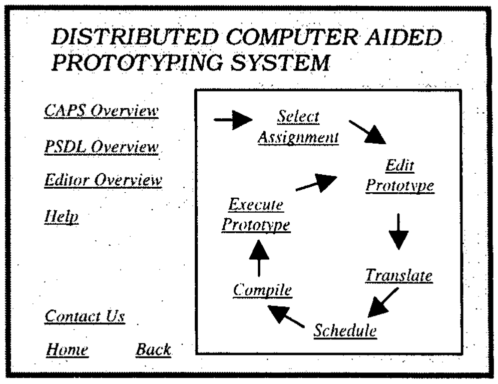

Figure 3. The DCAPS Web Interface

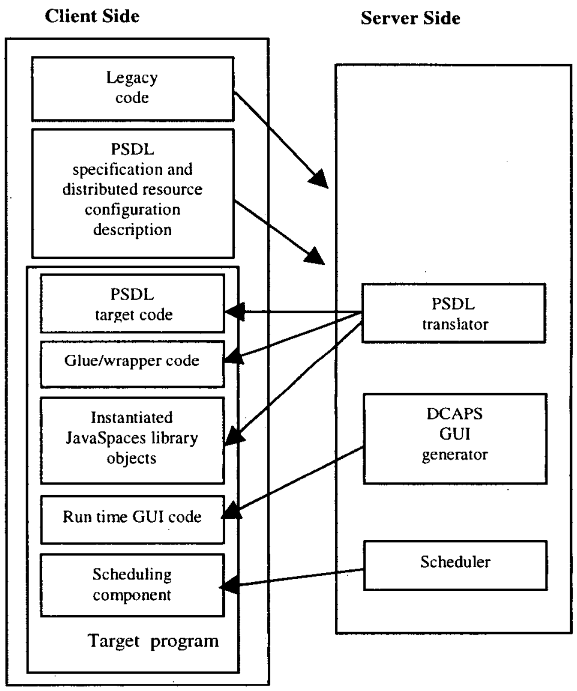

Figure 4. Compile Time Slice of Architecture

Several subsystems that generate source code at various levels are involved in PSDL compilation. The PSDL translator itself produces PSDL target code and wrapper and glue code to connect the PSDL target code to other distributed components. In this process the objects from the JavaSpaces library are instantiated and integrated with the target code. The DCAPS GUI generator produces run-time GUI code which serves as the user interface wrapper for the legacy system. Finally, the static scheduler automatically generates the schedule code component that ensures the target program observes the real-time constraints specified by the PSDL specification.

The existing PSDL data streams are encapsulated as generic Ada™ objects that provide the basic *read* and *write* operations. The actual behavior of the *read* and *write* operations varies depending on whether the data is a FIFO buffer or Sampled buffer. Such encapsulation makes the extension of PSDL data streams to JavaSpaces objects transparent. The only modification is to invoke the JavaSpaces *service registration* operation during the instantiation and initialization of the data objects, and to use the *read*, *write* and *take* JavaSpaces library operations to implement the *read* and *write* PSDL operations.

3.3 Run time slice of the architecture

The current principle of the DCAPS run time architecture is to delegate the inter-process communication layer and scheduling mechanism to the server side (Figure 5). The prototyping session starts at the client side by notifying the server and other clients (by remote login).

The process instances running on one or several client sites use wrappers and instantiated JavaSpaces library objects to send and receive messages. The JavaSpaces library via the underlying tuple space provides the environment for message flow between processes.

The server side also maintains the global logical clock used by the run time scheduler to synchronize process communication and to activate process instances according to PSDL semantics.

Another set of Java-based wrappers for user GUI's generated by DCAPS at compile time provides platform-independent process I/O. Execution traces, i.e. message transaction logs, could be created and stored at the server side for future analysis of the prototyping session.

3.3.1 Synchronization and Logical Clock

The formal real-time model of PSDL is based on the notion of a global clock [12][13]. When operators allocated to different hardware nodes must communicate within strict deadlines, we must account for network delays and imperfect clock synchronization. Our architecture uses local clocks and time reference signals that are broadcast once per iteration of a cyclic schedule to approximate a global clock. Each processor has one such schedule, and all schedules cover the same length of time. The time reference signals determine the local time for the beginning of the schedule at each node. Periodic re-calibration of these time references prevents divergence of the local clocks over long periods of time.

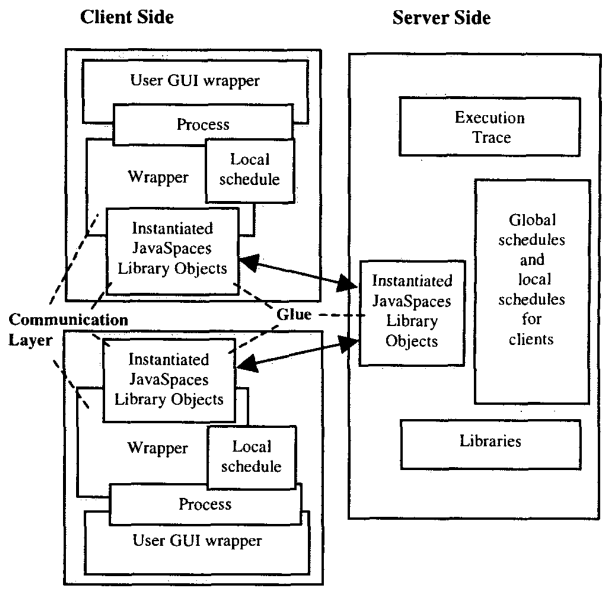
Client Side
Server Side
User GUI wrapper
Process
Wrapper
Local schedule
Instantiated JavaSpaces Library Objects
Communication Layer
Glue
Execution Trace
Global schedules and local schedules for clients
Instantiated JavaSpaces Library Objects
Libraries
Instantiated JavaSpaces Library Objects
Wrapper
Local schedule
Process
User GUI wrapper

Figure 5. Run Time Slice of Architecture

The worst-case point-to-point network delay bounds initial differences between local clocks. Hardware clocks with stable rates are available and relative drift rates are typically small. The product of the worst-case clock drift rate and the length of the schedule bounds clock drift error. The schedule must account for worst-case clock differences and worst-case clock drift error in addition to worst-case network latency between two nodes when scheduling two operations with a data flow precedence constraint [12][13].

3.3.2 Accurate Simulations on Imperfect Networks

Absolute guarantees of real-time constraints are clearly impossible when designers have no control over the network. In order to simulate a network with guaranteed real time service on an imperfect network, we need the notion of simulated time and supporting mechanisms in the form of:

- Time stamps attached to all communicated data values,
- A time-out period attached to every data communication to work around unbounded delays in the network,
- The mechanism for logical clock synchronization,
- Message buffering for sampled streams based on time-stamp order.

All this results in an accurate approximation of the behavior of a PSDL prototype on a target network with real time service guarantees in a prototyping environment whose networks have no such guarantees.

4 Current state and future work

A Java-based prototype editor has been implemented for the DCAPS. It has been tested in Windows NT, Linux, and Solaris™ environments. Different native interfaces have been implemented as the language wrappers for the Java Spaces-based communication library so that it can be called from applications implemented in different languages. Java Native Interface™ (JNI) makes the library available for C programs, while ActiveX™ wrappers enable Visual Basic™ (VB) programs to call the functions directly. The JNI wrapper makes it possible to create an interface between Ada and C so that programs in Ada can use JavaSpaces services.

The use of centralized control imposes extra communication overhead and creates potential bottleneck on the target heterogeneous system. We plan to conduct empirical studies to analyze the performance of such an approach in support of real-time systems, and investigate ways to relax centralized control by allowing bounded clock drifts among local clocks while still adhering to the constraints imposed by the PSDL timing model.

The current DCAPS scheduler generates a static assignment of the operators of the distributed prototype to the target network. In order to improve the global performance and efficiency of the distributed system, the runtime environment may require a dynamic scheduler to perform runtime load balance and operator reassignment. The mobility provided by the JavaSpaces-based library will support such requirement.

The DCAPS system provides a useful tool for distributed real-time software rapid prototyping in a distributed environment. The wrapper/glue method used in DCAPS can be generalized to system construction and interconnection of legacy systems. By automatically generating the codes for the "wrappers and glue" and providing a powerful environment, DCAPS allows the designers to concentrate on the interoperability problems and issues, freeing them from implementation details. It also enables easy reconfiguration of software and network properties to explore design alternatives. DCAPS is an on-going research project for the development and refinement of its prototyping tools.

References

[1] L. Bernstein, "Forward: Importance of Software Prototyping", *Journal of Systems Integration - Special Issue on Computer Aided Prototyping*, 6(1), pp. 9-14, 1996.

[2] Luqi and W. Royce, "Status report: computer-aided prototyping", *IEEE Software* , 9(6), Nov. 1992, pp. 77 -81.

[3] M. Boasson, *IEEE Software - Special Issue on Architecture*, 12(6), Nov 1995.

[4] S. Mellor and R. Johnson, *IEEE Software - Special Issue on Object Methods, Patterns, and Architectures*, 14(1), Jan/Feb 1997.

[5] Luqi, V. Berzins, "Rapidly prototyping real-time systems", *IEEE Software*, September 1988, pp. 25-36.

[6] Luqi and M. Ketabchi, "A computer-aided prototyping system", *IEEE Software* , 5(2), March 1988, pp. 66 -72.

[7] Luqi, "Computer-aided prototyping for a command-and-control system using CAPS", *IEEE Software*, 9(1), Jan. 1992, pp. 56 –67.

[8] Luqi, V. Berzins, R. Yeh, "A prototyping language for real time software", *IEEE Transactions on Software Engineering*, 14(10), October 1988, pp. 1409-1423.

[9] H. Cheng, *Automated generation of wrappers for interoperability*, Master's Thesis, Naval Postgraduate School, Monterey, Calif., March 2000.

[10] Luqi, M. Shing, "Real-time scheduling for software prototyping", *Journal of Systems Integration - Special Issue on Computer Aided Prototyping*, 6(1), pp. 41-72.

[11] J. Yang, and G. Kaiser, "JPernLite: Extensible Transaction Services for the WWW", *IEEE Transactions on Knowledge and Data Engineering*, 11(4), July/August 1999, pp. 639-657.

[12] Luqi, "Real-Time Constraints in a Rapid Prototyping Language", *Computer Languages*, 18, 1993, pp. 77-103.

[13] B. Krämer, Luqi and V. Berzins, "Compositional semantics of a real-time prototyping language", *IEEE Transaction of Software Engineering*, 19(5), May 1993, pp. 453-477.

[14] E. Freeman, S. Hupfer and K. Arnold, *JavaSpaces: Principles, Patterns, and Practice*, Addison-Wesley, 1999.

[15] D. Gelernter, "Generative Communication in Linda," *ACM Trans. Programming Languages and Systems*, 7(1), Jan. 1985, pp. 80-112.