

Port Monitor: A Monitoring & Debugging Approach For Component Frameworks

Torsten Wilde and James A. Kohl
Oak Ridge National Laboratory ¹

Keywords: components, component-based software engineering,
debugging, monitoring, CCA

Abstract. This paper discusses the concept of “port monitors” for debugging component-based software applications for the high performance Common Component Architecture (CCA) framework. CCA defines lightweight and less intrusive frameworks than non-scientific component based systems in order to maximize the application performance. CCA-compliant frameworks can be used across different application domains and have support for components written in a variety of languages used in scientific computing today. Because of that complexity component applications are hard to debug using traditional methods. Port monitoring is the idea of inserting a specialized port monitoring component between component interface ports. This Port Monitor component collects traces of all port invocations and transferred data between two components, taking advantage of the well-defined code separation in these components. By doing so it enables language-neutral debugging and allows the user to find and isolate faulty components, to simulate their invocation for specific debugging, to check port method data argument boundaries and to create files for black-box testing.

1. Introduction

Component based software design and development has become increasingly popular in the general computing world in the past decade. Many efforts focus on web-based and/or business software development, where high performance is not the main focus. Examples of such component-based frameworks are Microsoft.NET[1], Java Beans[2], Corba[3] and Microsoft DCOM[4]. In that environment frameworks are totally in control of the application execution and can more easily collect runtime data. In addition most commercial frameworks are developed, maintained and supported by one company, and most of them are tailored for a specific problem area or specific language. Because of this limited scope, debugging support is usually integrated in “developers” editions or sold as plug-ins. Unfortunately these frameworks don’t fit the requirements of the *scientific* community either because they do not run with HPC systems or languages or they run far too slow for there applications.

CCA[5] is an interdisciplinary effort to create a high-performance component framework for scientific computing. Here the focus shifts to lightweight frameworks that provide domain flexibility, language interoperability and high performance. In these environments, debugging of component-based applications becomes much harder. Many of the assumptions made in commercial frameworks are invalid here, and, therefore, a debugger has to handle the scenario of an application consisting of components compiled without debugging information, components written in different languages, large datasets, distributed parallel components, long application runtime and lightweight frameworks that don’t interfere with or control the overall application execution.

This extended abstract describes the challenges associated with debugging and error localization in such frameworks for scientific computing. The concept of port monitoring is introduced to address some of these challenges by logging component interface communication and the data arguments transferred between components. That information can then be used to isolate faulty components from the application, to analyze method calls (e.g. boundary checking), to simulate component invocation for debugging purposes, and to create test sets for component black-box testing.

¹ Research supported by the Mathematics, Information and Computational Sciences Office, Office of Advanced Scientific Computing research, U. S. Department of Energy, under contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

1.1. Common Component Architecture

The CCA was started in 1997 as an effort to bring the component programming model to scientific users. CCA is a specification of a component model and the interfaces for the required support structure or framework.

Component programming is an evolution of object-oriented programming. Components expose interface information publicly which can be dynamically discovered during runtime whereas most object-oriented languages define those properties statically at compile time. By hiding the component implementation behind standardized interfaces applications can be assembled by connecting component interfaces. More complex functionality can be encapsulated and reused this way.

The CCA component model uses the *provides-uses* design pattern. From the point of view of a component, there are two types of ports. Those that are implemented by a component are known as provides ports, and other components may connect to and use them. Other ports that a component will expect to be connected to and call are known as uses ports. Uses and provides ports are connected together as shown in Figure 1. The act of connecting components is referred to as component composition.

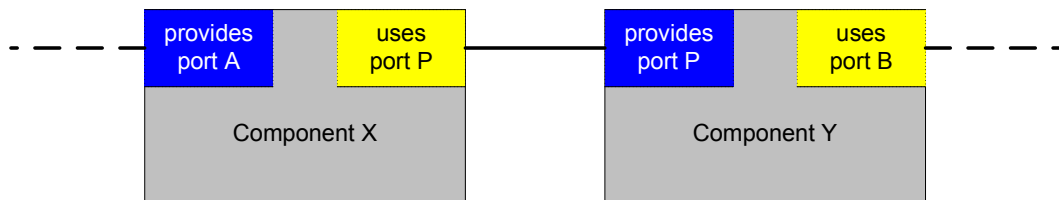


Figure 1: Two CCA components, component X uses “P” port provided by component Y

Components are peers and are independent. They are created and exist inside a framework; this is where they register themselves, declare their uses ports and provides ports and connect with other components. Port discovery is a service provided by the framework and is actually just another port that a component can connect to. For instance, a component can obtain a list from the framework of all components providing a specific interface or port. The component could then connect to each of the ports in the list in an iterative fashion and make calls on the methods within the connected port. To build a CCA application, an application developer simply composes together a set of components using a CCA-compliant framework. Details regarding the flexibility, performance and design characteristic of CCA applications can be found in [7].

Following is a list of used CCA terminology and concepts:

- Component: Software entities that communicate with the outside world via well defined, standardized interfaces, called ports, and that are the smallest visible building block for component applications
- Port: well defined communication connection interface between components
- Framework: simple container for holding, composing and executing component applications, components can't run without it

1.2. Component Application Debugging

One major difference between “traditional” and component application debugging is that component applications need a component framework to run and can be composed from third party components and components written in different languages. In addition a single component can't be easily executed independently of the component framework or other components because it relies on framework services and provide ports from other components in order to function. Here the traditional debugging tools, like source code debuggers, are of limited help. This is mainly because they assume boundaries which are softened or removed by the component paradigm. The key is to debug at the component interface level first where we can use higher level of information that is not available to standard source code debugging tools. Using that information hopefully allows splitting the application into easier to debug pieces. The three main interest areas for debugging and error checking at component application level are:

- Locating the failure and the components involved
- Understanding and visualizing the component application execution
- Checking and validating data values transferred between components

By providing support for this type of debugging we can ensure high quality and interoperability of components and can add high-level debugging support for component applications that use third party components and components written in different languages.

2. The Port Monitor

The port monitor component is similar to the proxy component used for capturing profiling data in CCA applications [6] in the sense that it also intercepts and forwards all port activity. But using the idea of port monitoring for debugging involves the control and validation of communication at port or interface level those requiring and storing much more information than just profiling data. By using post processing and analysis generated port interface tracefiles can be used to increase the level of information provided to the developer in order to assist in software debugging. Figure 2 shows the internal design organization of a Port Monitor component. The current port monitor prototype is written in C++ and has to be built by hand. That means the user has to build a port monitor for every connection he would like to monitor. In the future this should be done automatically without user involvement.

The port monitor logs port invocations and saves invocation and return information, like method parameter values, invocation results and return values. This execution event trace file can be used to support different functionality depending on user needs, namely:

- Replay and analysis of component behavior
- Data and control flow analyses of the application at component level
- Dependency and execution analysis of the application at component level
- Setting breakpoints on port method calls for interactive tracing
- Unit testing and evaluation with empirical data for single components
- Profiling for performance analysis, like method call counters, timings between specific method calls and durations of invocations
- Support for aspect-oriented computing by enabling the insertion of patterns at port level

Figure 3 shows an example component application with port monitoring components. After recording Component X port communication, X can be isolated from the application for replay and testing using only Port Monitor components.

3. Future Work

The idea of port monitoring presents many challenging research areas. The current work focused mainly on the development of a basic working prototype that stores CCA port communication information and provides an analyzing tool with very limited functionality.

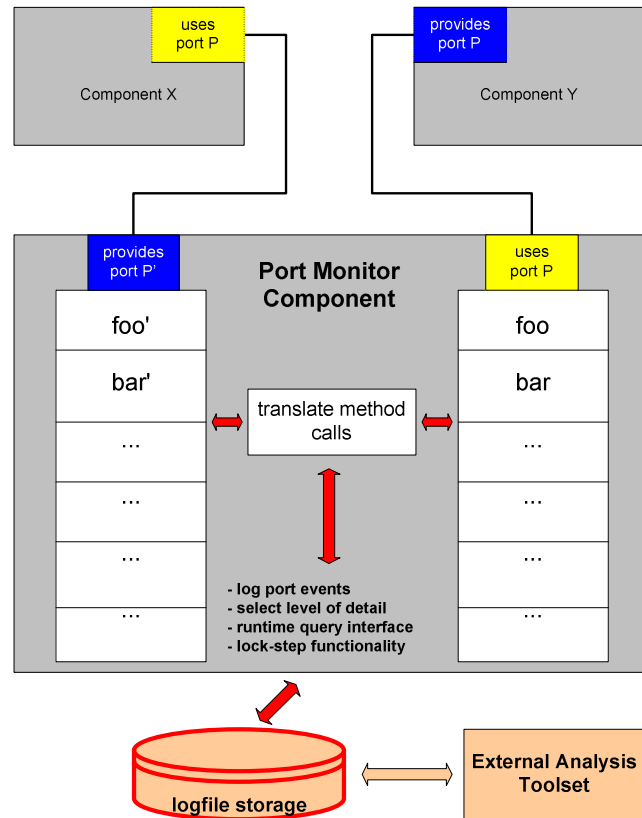


Figure 2: Port Monitor internal design

The near term goal is to provide the replay functionality in order to separate one component from the main application. This requires a lot of storage space for tracefiles which might not be available. Therefore the user should be able to select the required detail level for it's traces. He might choose to only collect statistics, like time spend in component port method calls or method counters, or he can save the execution history with or without transferred data values, or might choose to collect traces only for specific intervals, ports, port methods or port method data values. Another interesting trace might be only the execution event order which could be used for debugging parallel components.

Also the analyzing tool needs to grow in complexity and functionality to accommodate a wide range of possible options, like showing specialized traces from logfiles concerning only specific ports, port methods, port methods with specific parameter ranges, port methods where the parameters are below or above a specific value, e.g. useful for automatic boundary checking. This leads to the problem of how to store and access huge data arrays or complex data objects. Both need some sort of meta information in order for the port monitor component to save them, like serialization for objects and sizes and dimension information for arrays.

Another possibility for storing traces could be data bases which are optimized for complex queries.

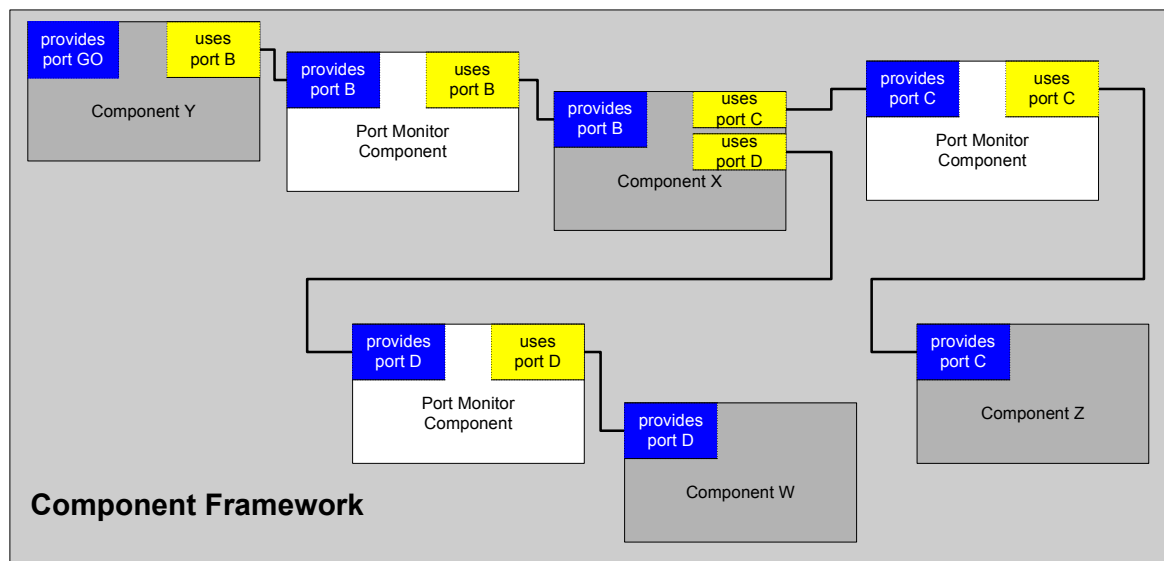


Figure 3: Logging component T's connections

References

- [1] Introducing Microsoft .Net, Third Edition by David S. Platt, Microsoft Press; 3rd edition (May 7, 2003)
- [2] D. Kara.: The Enterprise JavaBeans Component Model. Component Strategies 1(7) (1999)
- [3] CORBA Components, Object Management Group, OMG TC Document orbos/99-02-95. <http://www.omg.org>. (1999)
- [4] G. Eddon and H. Eddon: Inside Distributed COM. Microsoft Press (1998)
- [5] R.Armstrong, D.Gannon, A.Geist, K.Keahey, S.R.Kohn, L.McInnes, S.R.Parker, and B.A.Smolinski: Toward a Common Component Architecture for High-Performance Scientific Computing. In Proceedings of High Performance Distributed Computing Symposium, 1999
- [6] J.Ray, N.Trebon, R.C.Armstrong, S.Shende and A.Malony: Performance Measurement and Modeling of Component Applications in a High Performance Computing Environment: A case Study. In Proceedings of 18th International Parallel and Distributed Processing Symposium (IPDPS'04)
- [7] S.Lefantzi, J.Ray and H.N.Najm: Using the Common Component Architecture to Design High Performance Scientific Simulation Codes, In Proceedings of International Parallel and Distributed Processing Symposium, 2003