What is SINS?

Ramesh Bharadwaj
Center for High Assurance Computer Systems
Naval Research Laboratory
Washington, DC 20375 USA
ramesh@itd.nrl.navy.mil

Abstract

The goal of the NRL secure middleware project (SINS) [1, 2, 3] is to develop infrastructure for the deployment and protection of time- and mission-critical applications on a distributed computing platform, in a hostile computing environment such as the Internet, while using unreliable or untrusted COTS components. In this project, we are rethinking basic ways in which distributed applications are developed, deployed, configured, and maintained. Such a methodology and infrastructure are crucial for the deployment of mission-critical applications in a collaborative environment spanning multiple sites, by multiple agencies, and across multiple enterprises.

1. Introduction

Building robust and secure distributed systems in the presence of transient faults, node failures, and changes in network topology or connectivity poses a multitude of challenges: Software running on a distributed infrastructure must take into account changing situations such as network and node failures, changes in network topology and bandwidth, congestion, and latency. Also, the technology must be easy to use and reconfigure. Finally, the implementation must be verifiable. Today’s systems are built using highly reusable hardware and software components using the so called “system of systems” approach. Most systems being built today are the integration of highly disparate components that interact with one another via a middleware infrastructure. Some of these components may be Commercial Off The Shelf (COTS) or standard IP hardware components which may have been developed without taking into account the requirements of the system in which they are deployed. Further, during the design of a component, consideration of non-functional requirements such as reliability may complicate the design. Therefore, satisfying certain requirements of the system, such as fault tolerance, is better done at later stages of the development cycle during system integration.

Our methodology, language, and infrastructure are multi-faceted. It has been our experience that a simple description of our framework is inadequate to expose all these facets. Therefore, in order to explain our philosophy and approach, we use the fabled “blind men and the elephant” narrative style; i.e., we look at the problem and its solution from several perspectives, in the hope that this will enable the reader to piece together a more comprehensive picture.

2 Programming Methodology

The methodology of object-oriented programming provides for data abstraction (objects), but does not adequately address abstractions for or encapsulation of control flow. This problem is exacerbated in a distributed setting, where we have a “sea of objects” located at several sites on diverse hosts, and whose method calls need to be coordinated in a decentralized manner. The only mechanism used today for accomplishing this is the remote procedure call, which is executed from within the method of one object on a method of another. This clearly breaks encapsulation, leading to a spaghetti-like flow of control between the methods of each individual object. This is a problem that remains unsolved in service-oriented architectures, in particular web services, where an acceptable solution to the “distributed orchestration problem” has remained elusive.

In SINS, we separate concerns of data abstraction and control abstraction. We remain committed to the “sea of objects” and their methods, as in object-oriented programming; however, invoking the methods of various objects, be they local or remote, and coordinating these invocations, are the concern of software agents, which provide encapsulation of control flow. I.e., agents are to control as objects are to data. This view greatly simplifies object coordination and facilitates control to be distributed across platforms.

3 Networking

The separation of concerns related to data abstraction and control abstraction in our framework is similar to the signaling system architecture of telephone networks in which a separate, high-speed, out-of-band, packet-switched data communications network is used to control voice transmissions, which are conveyed on a separate circuit-switched network. A phone call cannot take place without this signaling framework. Similarly, our architecture can be used to control and coordinate the flow of real-time data, such as...
video and voice feeds, which are transmitted using conventional transport protocols such as TCP or IP.

4 Distributed Systems Programming

Any solution to the distributed coordination problem needs to be robust; application developers must not have to deal with race conditions, deadlocks, and other aspects of run-time behavior that are the bane of distributed systems development. I.e., behavior exhibited by an application during its execution on a single-threaded machine should be identical to its runs when fielded on a distributed platform. We achieve this in SINS by the use of the synchronous programming model, where one may assume that an external event is processed completely by an application (even if it is distributed across several machines) before the arrival of the next event. This greatly reduces the effort expended in debugging a distributed application, and explodes the myth that a distributed application must necessarily be non-deterministic.

5 Reconfigurable Middleware

Although commercial vendors spout the byline “middleware is everywhere” they only pay lip service to portability, platform independence, and simplicity. Current middleware is bloated, requires “big iron” to run on, and reduces usefulness to the lowest common denominator by trying to address every need of every application domain. Our solution to this “generality vs efficiency” conundrum is to make the middleware reconfigurable. In other words, application developers are given the flexibility to include services (and only the services they need) in the middleware, thereby reducing the overhead to a bare minimum.

6 Development Processes

Model-driven development (MDD) is an approach to software development in which comprehensive models are developed before source code is written. However, the research challenge of MDD is whether it is possible to automate the process, i.e., whether it is possible to automatically transform the models into code. In our approach, developers are given the flexibility of working at a high level of abstraction, e.g., using visual development tools, which are directly translated into a language that can be compiled and executed on a distributed platform. Whereas agile programming methods are code-centric, our approach instead focuses on developing high level artifacts of a high fidelity, which can be efficiently executed as code.

7 Artificial Intelligence

Autonomous agents are a hot area of research. Our agents are reactive, autonomous, and situation aware. They also exhibit a form of rudimentary learning, since they may essentially be viewed as “rule engines.”

8 Architecture Description Languages

Architectural Description Languages (ADLs) provide a notation for describing architectural designs. However, without a formal basis, they serve merely as documentation aids and cannot be analyzed to predict behavior of a composition of sub-systems. In our approach, the high-level models we create may be viewed as architectural frameworks that are analyzed, validated, verified, simulated, and eventually deployed onto a distributed infrastructure.

9 Service Oriented Architecture

Our framework provides for the discovery and composition of services, whose operations are coordinated in a distributed manner. Such an orchestration is performed in a location-transparent manner, to automatically deal with issues such as hardware overload, network congestion, and system or application faults. In addition to being survivable, our agent-based architecture provides for formally verified guarantees of applications’ compliance with their service level agreements (SLAs).

10 Dependability

Distributed programs written in our framework are amenable to fully automated static analysis techniques—such as automatic theorem proving using decision procedures, or model checking — to ensure compliance of a system with application specific requirements. Therefore, our framework provides for the development of applications with verifiable guarantees.

References