An Adaptive Software Approach to Intrusion Detection and Response

P. K. McKinley, S. M. Sadjadi and E. P. Kasten

Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824
{mckinley,masoud,kasten}@cse.msu.edu

Abstract

This paper proposes the use of programming language constructs to support adaptive self-monitoring and self-reporting software. The methods are particularly well-suited to wireless mobile devices, where limited resources may constrain the use of certain software audits. An adaptive software architecture is described that supports run-time transformations on software components, enabling them to report internal details on how they are being used to other parts of the system. Effectively, any component of the system can be turned into an "informer" at run time, and the nature of the reported information can be adapted dynamically based on changing conditions or directives from another authority, such as an intrusion detection system. A prototype implementation is described. The operation of the system is demonstrated through an experiment in which it detects and responds to a malicious host that multicasts "noise" packets to a wireless iPAQ handheld computer.

1 Introduction

Designing a highly trusted component-based software system requires that every component, in its turn, satisfies the security policy defined for the entire system. A good security framework needs to adapt to changing policies as well as respond to changing environmental conditions, including perceived threats to the system. To help ensure correct and consistent operation of the system, these changes should be reflected in the behavior of individual software components. A key part of any trusted system is the audit mechanism, which enables information to be collected for analysis, including both off-line post mortem processing as well as run-time checks that produce an immediate response from the system. In many cases, the information gleaned is fairly low-level, and its acquisition requires assistance of the operating system and possibly network components. In other cases, the nature of the threat may be specific to the application domain, and hence only the application (or a middleware layer working on its behalf) can recognize and report the errant behavior. In this paper we focus on the latter.

The traditional approach to solving the application-level auditing problem is applied at the time of development and inserts the security-related code directly in those software components deemed relevant. Obvious drawbacks to this method include: (1) the security policy and requirements need to be known at development time, and (2) maintaining application code that is tangled with dynamic security concerns is a non-trivial task. Recently, the research community has started to address this issue in a variety of ways, including the use of agent hierarchies [3], mobile agents [6, 15, 16], and compile-time weaving of security code into applications using aspect-oriented programming languages [19]. We contend that a system should also be able to insert security-related code into components *at run time*, thus enabling not only the monitoring of any part of the system on demand, but also the collection of information not necessarily anticipated at the time of development. The ability to reconfigure the security aspects of components at run time is especially relevant to mobile computing environments. In handheld and wearable systems, constant monitoring of all parameters of interest may be too expensive in terms of computing resources and memory requirements. Rather, certain security checks associated with a component should be loaded only as needed.

Our interest in adaptable security mechanisms arises from our work on RAPIDware, an ONR-sponsored research project that addresses the design and use of adaptive middleware for protection of critical infrastructures, such as command and control networks, electric power grids, and nuclear facilities. The RAPIDware project focuses on developing unified software technologies, based on rigorous software engineering principles, to support different dimensions of adaptability while preserving functional properties of the code. One of the target domains of the project is support for interactive collaboration in highly dynamic and heterogeneous environments, where users interact using handheld or wearable computers and wireless networks.

In this paper, we propose an adaptive software architecture that supports dynamic transformation of software components, enabling them to expose or report internal details on how they are being used to other parts of the system. Effectively, any component of an application can be turned into an "informer" at run time, and the nature of the reported information can be adapted dynamically based on changing conditions or directives from another authority, such as an intrusion detection system. The code that acts as the informer need not have been compiled into the component at the time of development. Indeed, it may have been developed later in response to a new type of attack or potential failure mode. The proposed methods enable such audit code to be loaded across the network and dynamically inserted into the running application. We envision that the ability to adapt the audit-related parts of a system at run time may be particularly helpful to the developers of intrusion detection systems for mobile computing environments.

The remainder of the paper is organized as follows. In Section 2, we review the main features of the Adaptive Java programming language, an extension to Java that is used in this study. Section 3 describes how Adaptive Java can be used to create an informer from any regular Java object, and how the informer can interact with other system components. Section 4 presents an example where we created an informer from a normal Java MulticastSocket and used it to detect anomalous behavior on the audio connection of an iPAQ handheld computer. Section 6 presents our conclusions and discusses future directions.

2 Adaptive Java and MetaSockets

In an earlier paper [7], we introduced Adaptive Java, an extension to Java that supports dynamic reconfiguration of software components. A compiler, ajc, converts Adaptive Java code into pure Java code. Adaptive Java programs are built using regular Java classes as well as *components*, which can be thought of as adaptable classes. The key programming concept in Adaptive Java is that each component offers three interfaces: one for performing normal imperative operations on the object (*computation*), one for observing internal behavior (*introspection*), and one for changing internal behavior (*intercession*). Operations in the computation dimension are referred to as *invocations*. Operations in the introspection dimension are called *refractions*: they offer only a partial view of internal structure and behavior and are not allowed to change the state or behavior of the component. Operations in the intercession dimension are called *transmutations*: they are used to modify the computational behavior of the component.

An existing Java class is converted into an adaptable component in two steps, as shown in Figure 1. First a *base-level* Adaptive Java component is constructed from the Java class through an operation called *absorption*, which uses the `absorbs` keyword. As part of the absorption process, mutable methods called *invocations* are created on the base-level component to expose the functionality of the absorbed class. For example, we might create a base-level socket component by absorbing a Java socket class. The base-level socket component might provide a customized interface for an application-specific service, such as audio or video streaming. Moreover, unneeded methods of the base class are occluded at this level. In the second step, *metafication* enables the creation of refractions and transmutations that operate on the base component. Meta components are defined using the `metafy` keyword. Refractions and transmutations embody adaptive logic, but are intended for defining *how* the base level can be inspected

and changed. The logic defining *why and when* these operations should be used is provided at other meta levels or by other components entirely. Continuing our socket example, the meta level enables us to define stream-specific refractions and transmutations for error control, fault tolerance, and security management.

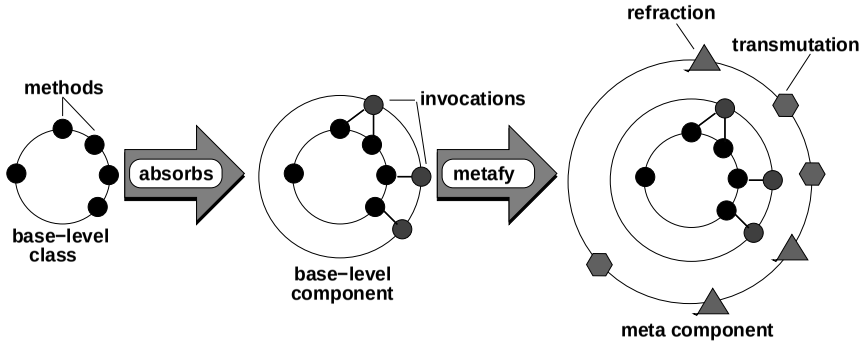

Figure 1: Component absorption and metafication in Adaptive Java.

We have used Adaptive Java to develop several adaptable components, including MetaSockets, which are constructed from the regular Java MulticastSocket class. The behavior of MetaSockets can be modified at run time in response to external conditions. For example, we used MetaSockets to enhance the quality of wireless connections by dynamically inserting code for forward error correction (FEC) coding and decoding. Figure 2 depicts the structure of a `MetaSocket` component that has been configured to perform two types of preprocessing, or *filtering*, on a data stream before it is actually sent using the internal Java Socket. The base-level component, called `SendSocket`, was created by absorbing the existing Java `MulticastSocket` class. Certain public members and methods are made accessible through invocations on `SendSocket`. This particular instantiation is intended to be used only for sending data, so the only invocations available to other components are `send()` and `close()`. Hence, the application code using the computational interface of a metamorphic socket looks similar to code that uses a regular socket. The `SendSocket` was metafied to create a meta-level component called `MetaSocket`. `GetStatus()` is a refraction that is used to obtain the current configuration of filters. `InsertFilter()` and `RemoveFilter()` are transmutations that are used to modify the set of filters that operate on the data stream.

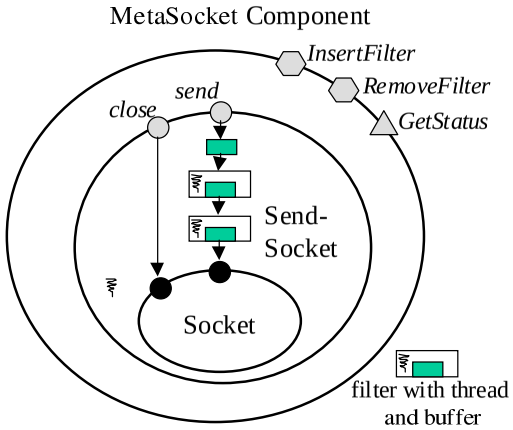

Figure 2: Structure of a MetaSocket implemented with Adaptive Java.

In the RAPIDware project, we are using Adaptive Java and MetaSockets for several purposes, including managing the quality-of-service of network connections, enhancing the fault tolerance of the system, and reducing the energy consumption of battery-powered devices. In this paper, we show how Adaptive Java can be used to support a security framework by dynamically turning arbitrary components into informer components.

3 An Informer-Based Architecture

In this section, we introduce the architecture of the proposed framework by way of a general example; a specific example and associated experiments are discussed in Section 4. Figure 3 illustrates the major parts of an adaptive, self-auditing software system written in Adaptive Java. In this example, most of the system executes on an iPAQ handheld computer. One program, called a Trader, executes on a desktop workstation. The two systems communicate via a wireless local area network. In Adaptive Java, each address space consists of one or more components, each of which in turn comprises several interacting components. The program running on the iPAQ in Figure 3 comprises six components: a Decision Maker (DM), an Information Event Mediator (IEM), a Component Loader (CL), and three application components, labeled simply A, B, and C. Components A and C are metafied composite components with a single level of composition, while component B has two levels of composition.

The system shown in Figure 3 contains three *informer* components that gather information of interest and make it available to other components. Any component developed in Adaptive Java can be transformed into an informer at run time, and no special consideration is needed when the component is initially defined. To do so, an Adaptive Java component that has been created through absorption is encapsulated with a meta-object protocol (MOP). The MOP provides refractions and transmutations that provide access to the internal state of the component. Other components can collect this information synchronously via refractions, or they can be notified asynchronously via events.

The creation of an informer is usually carried out by a *decision maker* (DM), an optional subcomponent within each component. According to a set of rules applied to the current situation, a DM controls all of the nonfunctional behavior of the the subcomponents of its container component. DMs are arranged hierarchically, and a DM inherits rules from a higher-level DM and might provide rules to lower-level DMs. In the example, the main component on the iPAQ contains a DM, called the root DM. Component A contains a DM, and component B contains two DMs one for each level of composition. When a DM is created, either upon system initialization or later during run time, it is instantiated with a default set of rules, some of which may later be overridden, removed, or modified by a higher-level DM. Depending on the rules and the current situation within its subsystem, a DM might decide to transform a particular component into an informer by metafying the component with a security-oriented MOP. A DM rule provides either a syntactic or semantic description of the MOP component, so the MOP need not reside on the system a priori. Instead, the DM contacts the *component loader* (one per address space) and requests the needed component. If the component loader does not find the component in its cache, it sends a request to a *Trader*. A Trader is a server, possibly running within the wired network, that returns a component implementation corresponding to a syntactic or semantic component request. The ability to dynamically load MOPs is especially important for mobile devices, where resources might be limited and overhead should be minimized.

A DM can invoke transmutations on an informer to start the logging of any part of the internal state of the component. Using this *polling-based* approach, however, an informer can only report information when explicitly requested to do so via a refraction. On the other hand, a *self-informer* component is an active component that, in addition to gathering information and making it available via refractions, also fires information events to notify the interested components called *listeners*. Self-informer components are needed when a situation demands immediate action. This paradigm, called *event-based* information propagation, is more efficient than polling when events are relatively infrequent but need to be handled immediately.

An *information event mediator* (IEM) is a unique component in the address space that decouples self-informers from the listeners. A listener can register its interests to specific information events with the IEM. All self-informers notify the IEM of any information event they observe by firing the corresponding event. The IEM, in its turn, notifies all the listeners interested to the events that just fired. Listeners handle the events appro-

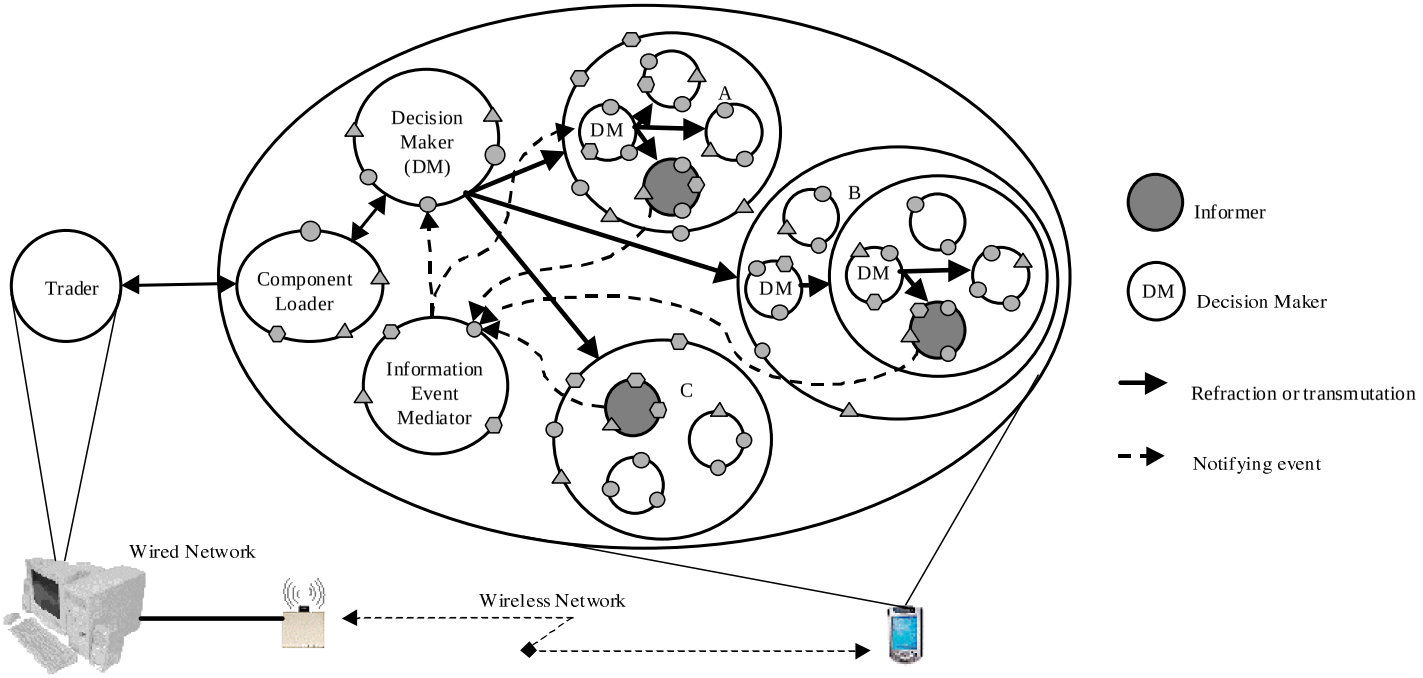

Figure 3: Example of self-monitoring mobile software system constructed with Adaptive Java.

priately afterwards. Typically, one of the listeners associated with events fired by informers will be a component of an intrusion detection system (IDS). The IDS component may contact other IDS components within the wired network for more intensive analysis, or may provide direct feedback to the DM based on local processing.

4 Example: Auditing Packet Stream Behavior

As described earlier, a MetaSocket is an adaptable component created from a Java MulticastSocket, by providing it with refractions and transmutations. Given the ubiquity of sockets in distributed applications and middleware systems, we use MetaSockets heavily in our studies of adaptable distributed systems. In particular, we explore how run-time adaptability of MetaSockets can be used to address different cross-cutting concerns,

such as quality-of-service, energy consumption, fault tolerance, and security. In this section, we describe the transformation of a MetaSocket into an informer that is used to detect anomalies in packet streams to mobile devices. Specifically, we investigated the use of informer sockets to monitor the behavior of wireless audio channels at run time.

Figure 4 shows the physical configuration of our experiments, where live audio is sent from a workstation to multiple iPAQ handheld computers running Windows CE. The audio stream is transmitted over a 100 Mbps Ethernet LAN to a wireless access point, where it is multicast on an 11 Mbps 802.11b wireless LAN. The audio streaming code comprises two main parts. On the sending station, the Recorder uses the `javax.sound` package to read audio data from a system's microphone and multicast it on the network. On the receiving station, the Player receives the audio data and plays it using `javax.sound`. Both applications were written in Adaptive Java and converted into pure Java using the ajc compiler described earlier. . They communicate using MetaSockets instead of regular Java sockets. The audio encoding uses a single channel with 8-bit samples. Relatively small packets are necessary for delivering audio data, in order to reduce jitter and minimize losses [13]. Hence, each packet contains 128 bytes, or 16 msec of audio. This interpacket delay at the sender (and implicitly, the delay between packets arriving at the receivers) stabilizes soon after the channel is established. Hence, any significant changes in this rate indicates either a malfunction or possible malicious behavior. In our experiment, we started a second source of packets to the multicast address used by the iPAQs, as shown in Figure 4, and used MetaSockets to detect and negate the effects of this source.

When the application begins execution, the DM for the MetaSocket at the receiver first calculates the initial expected rate of arriving packets, based on input parameters to the application. The DM then metafies the MetaSocket as an informer, creating a filter pipeline similar to that shown in Figure 2. The DM then inserts two filters at the receiving MetaSocket: the `NetArrivalRateMonitor` and `AppArrivalRateMonitor` are two self-informer filters that log the packet arrivals from the points of

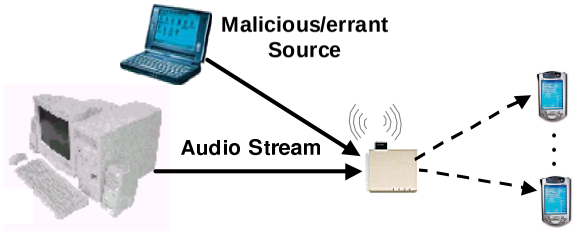

Figure 4: Physical experimental configuration.

view of the network and the application, respectively. These self-informer filters provide two MOPs. The first MOP is a refractive MOP that provides `isActive()`, `getExpectedInterArrivalTime()`, `getThreshold()`, and `getArrivalTimeVector()` methods. Respectively, these methods enable the DM to check if the self-informing option of the filter is active, retrieve the expected packet interarrival time set in the filter, retrieve the threshold percentage around which the packet interarrival time can vary and still be considered acceptable, and obtain the arrival time of the packet. The second MOP is a transmutative MOP that provides `setActive()`, `setExpectedInterArrivalTime()`, `setThresold()`, and `setArrivalTimeVector()` methods. Similarly, these methods can be used by the DM to activate or deactivate event generation by the filter, set the expected packet interarrival time, set the threshold percentage, and reset the packet arrival time. When initially inserted, the `NetArrivalRateMonitor` filter is deactivated and the `AppArrivalRateMonitor` is activated. Hence, the `AppArrivalRateMonitor` calculates the packet arrival rate while processing incoming packets and compares the current arrival rate against the `ExpectedArrivalRate`. If the current rate is higher than the `threshold` percentage over the `ExpectedArrivalRate`, then the filter fires a `HighArrivalRateEvent` event.

At a time 23 seconds into the experiment, we started the malicious source, which transmits a second audio stream (effectively noise packets) to the

receiver. Figure 5 shows the packet arrival rate from the network point of view calculated by the `NetArrivalRateMonitor`. After the malicious source starts (at point 23), the arrival rate goes from 60 packets per second to 120 packets per second.

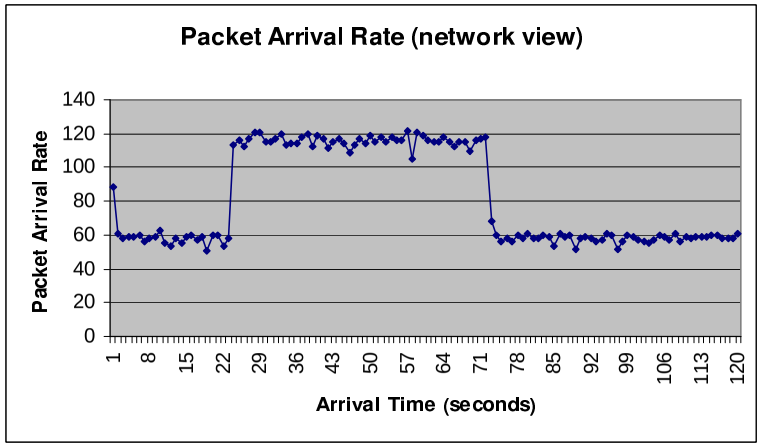

Figure 5: Trace of packet arrival rate per second from network point of view. A malicious source starts after 23 seconds and stops after 73 seconds.

Figure 6 shows the packet arrival rate observed by the application. The `AppArrivalRateMonitor` detects the high arrival rate and fires a `HighArrivalRateEvent` event. This event is handled by the DM by sending a request to the sending side's DM to insert a `DigestGen` filter, which is inserted at the first position in the sending side filter pipeline. The `DigestGen` filter calculates a digest for each outgoing packet using a secret key that is negotiated over the control channel between the sender and receiver using their public and private keys. Its peer, a `DigestVer` filter, will be automatically inserted at the receiving side filter pipeline after the arrival of the first packet wrapped by a header of `DigestVerID` type. This filter drops any packets that do not contain the proper digest. As shown in Figure 6, the MetaSockets and DMs respond very quickly in order to prevent the noise packets from being delivered to the application.

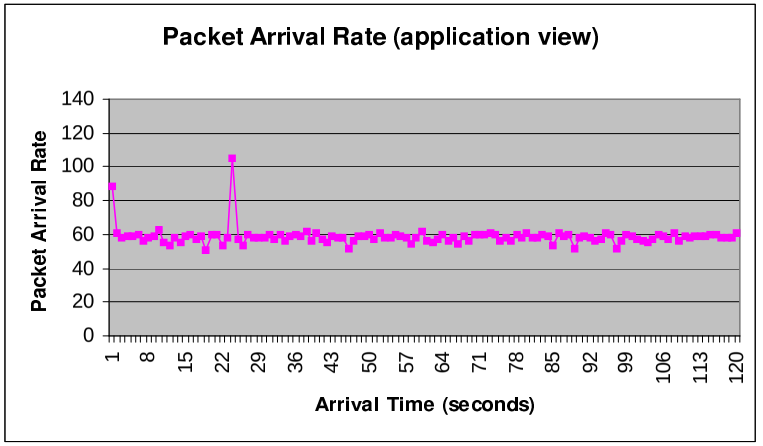

Figure 6: Trace of packet arrival rate per second from application point of view. The application sees only a single spike in the arrival rate at time 23, as subsequent noise packets are dropped.

Once the high arrival rate is detected by the `AppArrivalRateMonitor` filter, the DM deactivates this filter to prevent the generation of redundant high arrival rate events. The `AppArrivalRateMonitor` will continue logging the packet arrivals in an informer mode (as opposed to a self-informer mode). At the same time, the DM activates the `NetArrivalRateMonitor`, which operates similarly to the `AppArrivalRateMonitor`, except that it detects the return to a normal arrival rate.

As shown in Figure 5, the malicious source stop transmitting at time 73. The `NetArrivalRateMonitor` filter detects a normal arrival rate at this point and fires a `NormalArrivalRateEvent`. The DM will handle this

event by sending a request to the sender side to remove the `DigestGen` filter. After the `DigestGen` filter is removed from the sending side, the `DigestVer` filter will be automatically removed from the receiving side when the first packet without a `DigestVer` header arrives.

Figures 7 and 8 show the number of packet arrivals at the network and application over the course of our experiment. Figure 7 shows that the network at the receiving side of the application receives both the original packet and the noise packet. Figure 8, on the other hand, shows that the application receives packets at a constant rate, since the additional noise packets have been dropped by the digest filters.

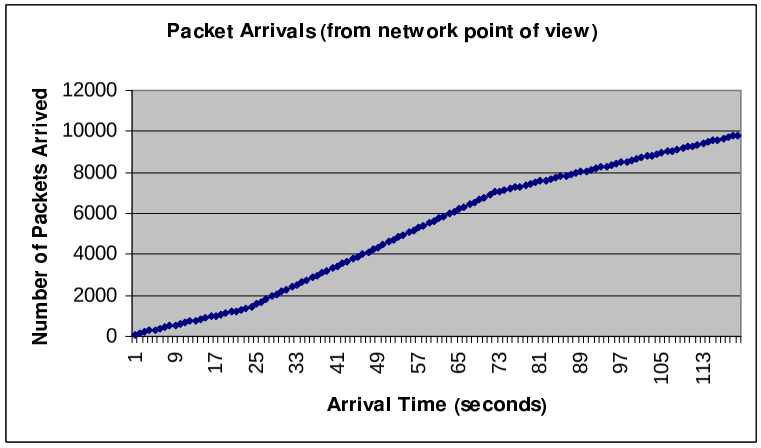

Figure 7: Trace of packet arrivals per second from network point of view.

The reader might wonder why a developer would write MetaSocket code that could deliver packets out of order to the application. Moreover, why do we need such a complicated framework simply to filter noise packets from a data channel that exhibits well-defined properties? Actually, these questions highlight precisely the advantages of an adaptive run-time auditing framework. Prior to metafication, the base-level socket component is simply responsible for receiving packets and delivering them to the application. It does not, and should not, care about application-level semantics, including sequence numbers. Moreover, the original developer might not have anticipated all possible audits on that component. In aspect-oriented programming terminology [8], the cross-cutting security code has been "detangled" from the code that implements the functional behavior of the component. When the MetaSocket is created through metafication, some basic checks are inserted into the component, specific to its use for audio streaming. However, to further inspect the data in the packet, including the application-level header, is not necessarily warranted and would likely waste computing resources. The proposed techniques enable all auditing functionality to be completely tailorable to current conditions and added only as needed.

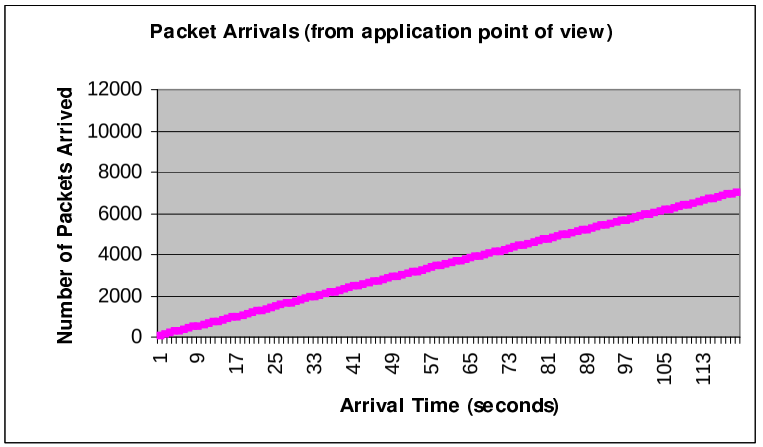

Figure 8: Trace of packet arrivals per second from application point of view.

5 Related Work

In recent years, numerous research groups have addressed the issue of adaptive middleware frameworks that can accommodate dynamic, heterogeneous infrastructures. Examples include MOST [5], TAO [10], dynamicTAO [9], Odyssey [14], MCF [11], and QuO [18]. In addition, many of these projects, as well as other middleware projects [16, 17, 20] provide support for adaptive security management, including auditing operations. Like RAPIDware, several of these frameworks involve computational reflection [12], which refers to the ability of a computational process to reason about (and possibly alter) its own behavior. However, most of these approaches operate above the programming language level, for example, in CORBA contexts. To our knowledge, none supports the capability described in this paper, namely, the run-time transformation of language-level components into informers. To provide this functionality, we take advantage of Adaptive Java's support for run-time adaptation. In this sense, our work complements many of the projects above: implementing parts of those systems in Adaptive Java would provide an additional level of adaptability in areas such as auditing. Moreover, these techniques facilitate the run-time adaptation of the system in ways not anticipated during the original development. While other researchers are investigating the use of programming language constructs to realize adaptable behavior [1,2,4], to our knowledge, those approaches have not been applied to the adaptive auditing problem.

6 Conclusions

In this study, we investigated the use of the Adaptive Java programming language to support run time adaptation of security-oriented audits. By providing mechanisms that enable monitoring and reporting code to be inserted into running components, any component can become an informer as needed, and the operation of the informer can be adapted to changing conditions and situations. While our prototype and the experiments conducted are relatively simple, they serve as a proof-of-concept that it is possible to insert auditing code into arbitrary components at run time so as to detect and respond to events of interest. We emphasize that this method can be applied to *any* part of the system, not only communication primitives.

We believe that this approach may be particularly useful in mobile computing environments, where constant monitoring of all parameters of interest may be too expensive in terms of computing resources and memory requirements. More importantly, how the system responds to a potential security threat may depend on several factors, such as available battery power and current network channel conditions. Indeed, a major goal of the RAPIDware project is to explore software mechanisms that enable coordinated adaptation to changing conditions in multiple cross-cutting concerns: security, energy consumption, fault tolerance, and quality of service. Currently, we are conducting subprojects in the use of Adaptive Java to address other key areas where software adaptability is needed in mobile devices: dynamically changing the fault tolerance properties of components; adaptive quality-of-service for audio, video, and data; mitigation of the heterogeneity of system display characteristics; and energy management strategies for battery-powered devices.

Further Information. Related papers of the Software Engineering and Network Systems Laboratory can be found at the following URL: `http://www.cse.msu.edu/sens`. The RAPIDware project homepage is `http://www.cse.msu.edu/rapidware`.

Acknowledgements. The authors would like to thank Udiyan Padmanabhan and Kurt Stirewalt for their contributions to this work. This work was supported in part by the U.S. Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744. This work was also supported in part by U.S. National Science Foundation grants CDA-9617310, NCR-9706285, CCR-9912407, EIA-0000433, and EIA-0130724.

References

[1] Adve, V., Lam, V. V., and Ensink, B. Language and compiler support for adaptive distributed applications. In *Proceedings of the ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems (OM 2001)* (Snowbird, Utah, June 2001).

[2] Akkai, F., Bader, A., and Elrad, T. Dynamic weaving for building reconfigurable software systems. In *Proceedings of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems* (Tampa Bay, Florida, October 2001).

[3] Balasubramaniyan, J. S., Garcia-Fernandez, J. O., Isacoff, D., Spafford, E., and Zamboni, D. Aspects for run-time component integration. In *Proceedings of the 14th Annual Computer Security Applications Conference* (December 1998), pp. 13–24.

[4] Bergmans, L., and Aksit, M. Composing crosscutting concerns using composition filters. *Communications of the ACM 44*, 10 (October 2001), 51–57.

[5] Friday, A., Davies, N., Blair, G., and Cheverst, K. Developing adaptive applications: The MOST experience. *Journal of Integrated Computer-Aided Engineering 6*, 2 (1999), 143–157.

[6] Jansen, W., Mell, P., Karygiannis, T., and Marks, D. Mobile agents in intrusion detection and response. In *Proceedings of the 12th Annual Canadian Information Technology Security Symposium* (Ottawa, Canada, 2000).

[7] Kasten, E., McKinley, P. K., Sadjadi, S., and Stirewalt, R. Separating introspection and intercession in metamorphic distributed systems. In *Proceedings of the IEEE Workshop on Aspect-Oriented Programming for Distributed Computing (with ICDCS'02)* (Vienna, Austia, July 2002). to appear.

[8] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (June 1997), Springer-Verlag LNCS 1241.

[9] Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., aes, L. C. M., and Campbell, R. H. Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2000)* (New York, April 2000).

[10] Kuhns, F., O'Ryan, C., Schmidt, D. C., Othman, O., and Parsons, J. The design and performance of a pluggable protocols framework for object request broker middleware. In *Proceedings of the IFIP Sixth International Workshop on Protocols For High-Speed Networks (PfHSN '99)* (Salem, Massachusetts, August 1998).

[11] Li, B., and Nahrstedt, K. A control-based middleware framework for quality of service adaptations. *IEEE Journal of Selected Areas in Communications 17*, 9 (September 1999).

[12] Maes, P. Concepts and experiments in computational reflection. In *Proceedings of the ACM Conference on Object-Oriented Languages (OOPSLA)* (dec 1987).

[13] McKinley, P. K., Padmanabhan, U. I., and Ancha, N. Experiments in composing proxy audio services for mobile users. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)* (Heidelberg, Germany, November 2001), pp. 99–120.

[14] Noble, B. D., and Satyanarayanan, M. Experience with adaptive mobile applications in Odyssey. *Mobile Networks and Applications 4* (1999), 245–254.

[15] Qian, T., and Campbell, R. Dynamic agent-based security architecture for mobile computers. In *Proceedings of the International Conference on Parallel and Distributed Computing and Networks (PDCN'98)* (December 1998).

[16] Tripathi, A., Ahmed, T., Pathak, S., Pathak, A., Carney, M., Koka, M., and Dokas, P. Active monitoring of network systems using mobile agents. In *Proceedings of Networks 2002, Joint Conference of ICWLHN 2002 and ICN 2002* (August 2002). to appear.

[17] Truyen, E., Jörgensen, B. N., Joosen, W., and Verbaeten, P. Aspects for run-time component integration. In *Proceedings of the ECOOP 2000 Workshop on Aspects and Dimensions of Concerns* (Sophia Antipolis and Cannes, France, 2000).

[18] Vanegas, R., Zinky, J. A., Loyall, J. P., Karr, D. A., Schantz, R. E., and Bakken, D. E. QuO's runtime support for quality of service in distributed objects. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)* (The Lake District, England, September 1998).

[19] Viega, J., and Evans, D. Separation of concerns for security. In *Proceedings of the ICSE Workshop on Multidimensional Separation of Concerns in Software Engineering* (June 2000).

[20] Yu, W., Xuan, D., Graham, B., Santhanam, S., Bettati, R., and Zhao, W. An integrated middleware-based solution for supporting secured dynamic-coalition applications in heterogeneous environments. In *Proceedings of the 2nd IEEE SMC Information Assurance Workshop* (United States Military Academy, West Point, New York, June 2001).

Philip K. McKinley received the B.S. degree in mathematics and computer science from Iowa State University in 1982, the M.S. degree in computer science from Purdue University in 1983, and the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign in 1989. Dr. McKinley is currently a Professor in the Department of Computer Science at Michigan State University, where he has been on the faculty since 1990. He was a member of technical staff at Bell Laboratories in Naperville, Illinois from 1982-1990, on leave of absence 1985-1989. Dr. McKinley serves as an Associate Editor for *IEEE Transactions on Parallel and Distributed Systems* and is co-chair of the program committee for the 2003 IEEE International Conference on Distributed Computing Systems. His current research interests include adaptive middleware, collaborative applications, mobile computing, and group communication protocols. He is a member of the IEEE and ACM. He may be contacted at mckinley@cse.msu.edu.

Seyed Masoud Sadjadi received the B.S. degree in hardware computer engineering from University of Tehran in 1995 and the M.S. degree in software computer engineering from Azad Tehran University in 1999. Sadjadi is currently a doctoral student in the Department of Computer Science and Engineering at Michigan State University. He worked in the software industry as a programmer, designer, and project manager from 1991-2000. His current research interests include adaptive software, middleware, collaborative applications, mobile computing, and multimedia applications. He is a member of the IEEE. He may be contacted at masoud@cse.msu.edu.

Eric P. Kasten received the B.S degree in mathematics and computer science from Central Michigan University in 1989 and the M.S. degree in computer science from Michigan State University in 1997. He is currently a doctoral student in the Department of Computer Science and Engineering at Michigan State University and a software developer for the National Superconducting Cyclotron Laboratory. His current research interests include adaptive middleware, collaborative applications, mobile computing and dynamic system composition. He is a member of the IEEE, ACM and the International Society for Adaptive Behavior. He may be contacted at kasten@cse.msu.edu.