Exploring the Role of Software Architecture in Dynamic and Fault Tolerant Pervasive Systems

Chiyoung Seo[1], Sam Malek[1], George Edwards[1], Daniel Popescu[1], Nenad Medvidovic[1], Brad Petrus[2], and Sharmila Ravula[3]

[1]*Computer Science Department*
Univ. of Southern California
Los Angeles, CA 90089, U.S.A.
{cseo,malek,gedwards, dpopescu,neno}*@usc.edu*

[2]*Bosch Rsrch & Tech. Center*
Two NorthShore Center, Suite 320
Pittsburgh, PA 15212, U.S.A.
brad.petrus@rtc.bosch.com

[3]*Bosch Rsrch & Tech. Center*
4009 Miranda Avenue
Palo Alto, CA 94304, U.S.A.
sharmila.ravula@rtc.bosch.com

Abstract

Pervasive systems are rapidly growing in size, complexity, distribution, and heterogeneity. As a result, the traditional practice of developing one-off embedded applications that are often rigid and unmanageable is no longer acceptable. This is particularly evident in a growing class of mobile and dynamic pervasive systems that are highly unpredictable, and thus require flexible and adaptable software support. At the same time, many of these applications are mission critical and have stringent fault tolerance requirements. In this paper, we argue that an effective approach to developing software systems in this domain is to employ the principles of software architecture. We discuss the design and implementation of facilities we have provided in a tool-suite targeted for architecture-based development of fault tolerant pervasive systems.

1. Introduction

The past few decades have witnessed an unrelenting pattern of growth in the size and complexity of software systems, which will likely continue well into the foreseeable future. This pattern is further evident in an emerging class of embedded and pervasive software systems that are growing in popularity due to increases in the speed and capacity of hardware, a decrease in its cost, the emergence of wireless ad hoc networks, the proliferation of sensors and handheld computing devices, and so on. Previous studies have shown that a promising approach to resolving the challenges of developing large-scale software systems is to employ the principles of software architectures [5,20]. Software architectures provide abstractions for representing the structure, behavior, and key properties of a software system [15,18]. They are described in terms of software components (computational elements), connectors (interaction elements), and their configurations. Software architectural styles (e.g., publish-subscribe, peer-to-peer, pipe-and-filter, client-server) further refine the vocabulary of component and connector types and propose a set of constraints on how instances of those types may be combined in a system.

For software architectural models to be truly useful in a development setting, they must be accompanied by support for their implementation [10][19]. This is particularly important in the context of pervasive systems: they are often complex, highly distributed, decentralized, heterogeneous, mobile, and long-lived, increasing the risk of architectural drift [15] unless there is a clear relationship between the architecture and its implementation. This suggests that state-of-the-art middleware solutions (e.g., CORBA Orbix, TAO) that lack the implementation-level facilities for key elements of software architecture (e.g., explicit support for software connectors or architectural styles) are not necessarily the best candidates for architecture-based software development.

This paper builds on our previous position [7,8], which has emerged from close to ten years of experience with embedded and pervasive environments: we argue that an *architectural middleware* platform, which provides native implementation-level support for the key architectural abstractions, is better suited than traditional middleware platforms to address the software engineering challenges inherent in developing pervasive systems. In this paper, we focus on the challenges posed by the growing class of safety- or mission-critical pervasive systems, for which fault tolerance is essential. Three key facilities are required for the development of fault tolerant pervasive systems: (1) dynamic discovery of new services and resources, (2) automated and transparent recovery from failure, and (3) analytical determination of component replication strategies and deployment architectures. We show that by making fault tolerance concerns explicit in the architectural models, we can analyze and improve the system's resilience to unexpected failures. Moreover, we demonstrate that an architectural middleware can provide advanced fault tolerance facilities, while achieving efficiency and architectural clarity. In support of the above arguments, we discuss our experience with a family of fault tolerant sensor network applications that has been developed as part of an ongoing collaborative project between the University of Southern California and the Bosch Research and Technology Center.

The remainder of the paper is organized as follows. Section 2 provides an overview of a family of sensor network applications that illustrates many of the concepts in this paper. Section 3 presents a middleware platform we have developed that implements an architecture-based solution to fault tolerance in pervasive systems. Section 4 discusses our support for service discovery. Section 5 presents the facilities for recovering from failure. Section 6 presents a tool-suite for exploring and analyzing a system's QoS properties, including its fault tolerance. Section 7 presents the related work. Finally, the paper concludes with a brief discussion and overview of future work.

2. Application Scenario

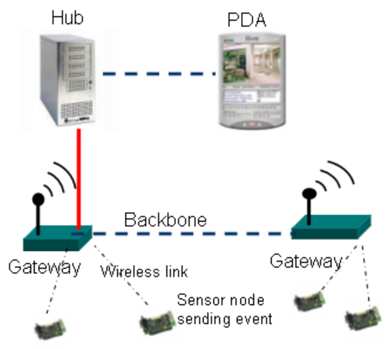

Figure 1. MIDAS system.

In this section, we describe a family of sensor network applications, called MIDAS. MIDAS is composed of a large number of sensors, gateways, hubs, and PDAs that are connected wirelessly in the manner shown in Figure 1. The sensors, which are used to monitor the environment around them, communicate their status to one another and to the gateways. The gateway computers are responsible for managing and coordinating the sensors. In addition, the gateways translate, aggregate, and fuse the data received from the sensors, and propagate the appropriate data (e.g., events) to the hubs. Furthermore, each gateway provides various services that can be used by other gateway hosts. For example, a gateway can provide a Global Logging Service that records events received from all the sensors by other gateways. Hubs are used to evaluate and visualize the sensor data for the users, as well as provide an interface through which the user can send control commands to the various sensors and gateways in the system. Hubs may also be configured to propagate the appropriate sensor data to PDAs, which are then used by the mobile users of the system.

Many instances of MIDAS operate in safety-critical or regulated environments, in which a given event must be delivered to its recipients under all conditions (e.g., medical emergency). However, many factors may impact MIDAS's ability to function correctly. For example, since the computing platforms (e.g., PDA, gateways, and sensors) have finite battery lives, a host may go down due to battery depletion. Similarly, the wireless communication among various MIDAS platforms is susceptible to permanent or temporary disconnections. Consequently, all the services provided by the components of a disconnected/depleted host become unavailable to other hosts. Therefore, the software system should support autonomic fail-over: if a host providing a given service fails, then another host in the system should quickly be enabled as a new provider of the same service.[1] Often times it is required for the computational state of the newly enabled service to be the same as the last known state of the failed service.

One approach to providing fail-over support is through the replication of software components, and thus the services they provide. However, component replication in MIDAS is difficult mainly due to its resource-constrained nature (i.e., limited CPU, memory, battery, network bandwidth, etc.). The existing approaches for providing fault tolerance on traditional desktop platforms are often inefficient in this domain. On top of this, given finite computing resources, components must be replicated selectively in order to achieve the maximum benefit. At the same time, while replication can improve fault tolerance, other QoS properties can degrade due to the overhead of executing additional components and keeping replicas synchronized. Finally, often the implementation complexity of advanced fail-over support (i.e., replication, synchronization, and recovery) and its coupling with the application logic, overwhelms the engineers. In the remainder of this paper, we discuss how we addressed these problems in the MIDAS project by employing the principles of software architecture. We believe that the techniques we have applied and the approach we have taken are applicable to pervasive applications in general.

3. Architectural Middleware

Prism-MW [7] is a middleware platform that supports architectural abstractions by providing implementation-level modules for representing and manipulating each architectural element. These abstractions enable direct mapping between an architecture and its implementation. Figure 2 shows the class design view of Prism-MW. The shaded classes constitute the middleware *core*, a minimal subset of Prism-MW that enables implementation and execution of architectures in a single address space.

Brick represents the different architectural building blocks that are described next. *Architecture* records the configuration of its constituent components, connectors, and ports, and provides facilities for their addition and removal. *Event*s are used to capture communication in an architecture. An event consists of a name and payload. An event type is either a *request* for a recipient component to perform an operation or a *reply* that a sender component has performed an operation. *Port*s are the loci of interaction in an architecture. A link between two ports is made by *weld*ing them together. Each Port has a type, which is either *request* or *reply*. An event placed on one port is forwarded to the port linked to it in the following manner: request events are for-

1. In the context of MIDAS a software component usually provides a number of services, which are typically made available to other components via the component's public interfaces.

warded from request ports to reply ports, while reply events are forwarded in the opposite direction. *Component*s perform computations in an architecture and may maintain their own internal state. *Connector*s are used to control the routing of events among the attached components. Connectors may support arbitrary event delivery semantics (e.g., unicast, multicast, broadcast). A component or a connector can have any number of attached ports. In order to support the needs of dynamically changing applications, each Prism-MW component or connector is capable of adding or removing ports at runtime. This property, coupled with event-based interaction, has proven to be highly effective for addressing system re-configurability.

Prism-MW's design is intended to be highly extensible while keeping Prism-MW's core unchanged. To that end, the core constructs (e.g., *Component*, *Connector*) are subclassed via specialized classes (e.g., *ExtensibleComponent*, *ExtensibleConnector*), each of which has a reference to a number of abstract classes (shown in Figure 2). Each abstract class can have multiple implementations, thus enabling selection of the desired functionality inside each instance of a given extensible class.

We have employed several novel optimization techniques to minimize the computation, communication, and memory overhead of the middleware [7]. For example, we have measured the memory overhead of Prism-MW's core to be less than 2.3KB. However, in the context of the MIDAS project we came to realize that for these architectural facilities to be truly useful in a highly heterogeneous and resource constrained environment, they would need to be complemented with the appropriate low-level system support. Furthermore, it became clear that to fully reap the benefits of developing a software system using the architectural facilities provided by Prism-MW, the middleware should be accompanied with several more advanced facilities. Our

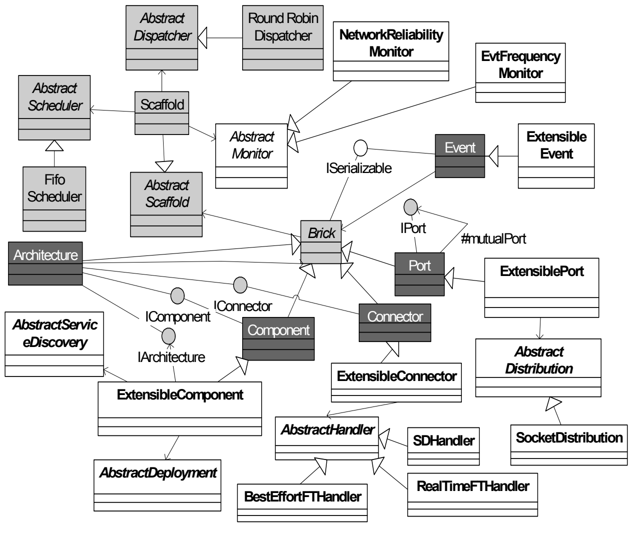

Figure 2. UML class diagram of Prism-MW. Middleware core classes are highlighted.

experience has shown that an architectural middleware for pervasive computing is composed of three distinct layers which are deployed on top of an OS (shown in Figure 3): at the bottom is a Modular Virtual Machine (MVM) layer that allows the middleware to be deployed on heterogeneous platforms efficiently; the abstraction facilities provided by the virtual machine are leveraged by the middleware's architectural constructs that lay on top of it; finally, these architectural constructs are leveraged to implement various pervasive computing facilities. In this paper we focus on several advanced fault tolerant facilities we have implemented by leveraging Prism-MW's architectural constructs. We point the interested reader to [8] for a more detailed discussion of the synergy between MVM and the architectural layer that has allowed us to satisfy the heterogeneity, efficiency, and scalability requirements in this domain.

4. Service Discovery

Service discovery in the embedded and pervasive environments is a challenging problem: (1) we typically do not know the location of a service provider at design-time; (2) a service provider may become unavailable due to hardware, software, or network failures; and (3) the location of a service provider may change at runtime (e.g., redeployment of a service provider). In this section, we discuss how we have implemented dynamic service discovery on top of Prism-MW's architectural constructs.

Figure 4 shows a small fragment of the MIDAS architecture that we use to illustrate some of the concepts in this paper. There are two types of components: service providers and clients. Service providing components could in turn provide either local or global services. For example, the Global Logging Service component running on Gateway 2 records and maintains event messages received from all the sensors of all the gateways. Often times a local service provider may need to invoke a global service provider to access information or services that are not provided locally. Therefore, local service providers have to be able to discover the global ser-

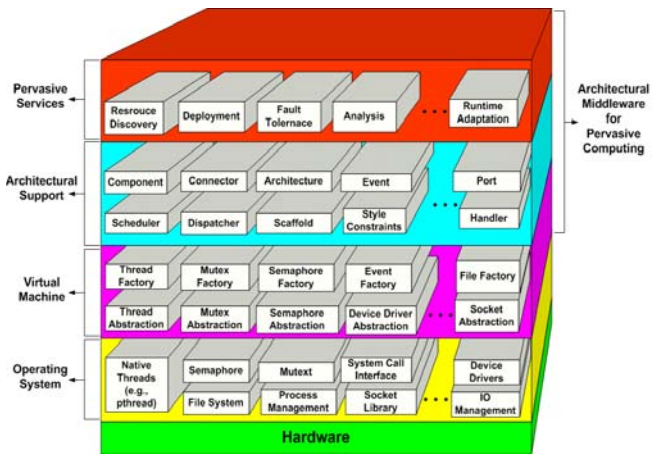

Figure 3. Layers of System Stack.

vice provider at runtime. As will be discussed in more detail in Section 5, fault recovery facilities in this domain also depend on the ability to discover services at runtime.

To support dynamic service discovery, we leveraged Prism-MW's extensible design. We developed an implementation of *AbstractServiceDiscovery* shown in Figure 2 that provides the support for recording and retrieval of the information about services. An *ExtensibleComponent* with an implementation of *AbstractServiceDiscovery* installed on it is called *SDEngine* (shown in Figure 4). A *Local SDEngine* acts as a service discovery agent on its local host. It is "architecturally aware": it can access its architecture to determine the services installed locally and maintain the database about these services. Each *Local SDEngine* transfers the information about all the services running on its local host to the *Global SDEngine* that manages the global database containing the information about the services running on all of the gateway hosts.

We also have developed an implementation of *AbstractHander* (shown in Figure 2), called *SDHandler*, which is responsible for routing a service request event to the corresponding service provider. An *ExtensibleConnector* with the *SDHandler* is called *SDConnector* as shown in Figure 4. In our solution, service lookup and invocation is performed by the interactions between *SDConnector*s and *Local/Global SDEngine*s. For example, suppose Client 3 on Gateway 3 receives an event from a sensor and wants to record the event in a persistent storage via the Global Logging Service running on Gateway 2. Client 3 then sends the event to its local *SDConnector*. However, since *SDConnector* does not have any location information about the Global Logging Service in its routing table, it sends a lookup request to the *Local SDEngine*, which then forwards the request to the *Global SDEngine*. The *Global SDEngine* retrieves the

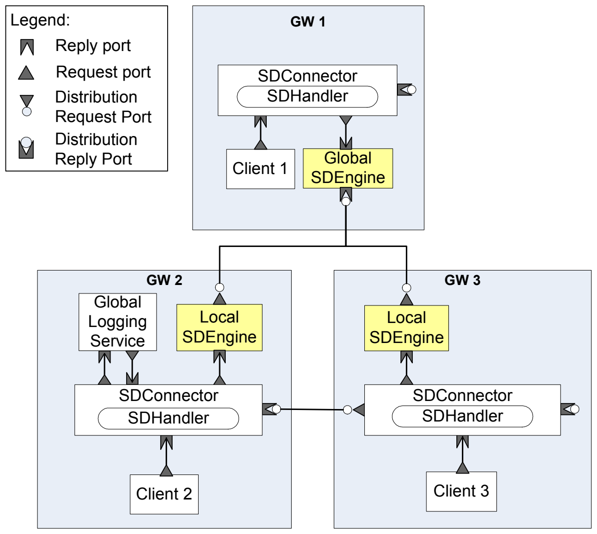

Figure 4. Service discovery support in the MIDAS system.

location information about the Global Logging Service and send it to the Gateway 3's *Local SDEngine*, which then records this location information in its database, and connects its local *SDConnector* to Gateway 2's *SDConnector* via distribution ports (i.e., instances of an *ExtensiblePort* with the *SocketDistribution* in Figure 2) as shown in Figure 4. After the above steps are performed, Gateway 3's *SDConnector* sends the request event received from Client 3 to the Global Logging Service via the connection with Gateway 2's *SDConnector*. Future invocations of the Global Logging Service by Client 3 (or other clients on Gateway 3) do not require the above service lookup steps.

As mentioned earlier (and will be further discussed in Section 6) the initial location of a service provider may change at runtime. If a component's location changes, the *Local SDEngine* running on the component's new host informs the *Global SDEngine* of this location change. The *Global SDEngine* will then broadcast the component's new location to all of the *Local SDEngine*s so that they can update their databases and make a new connection on their local *SDConnector*s with the component's new host.

There are several advantages to providing service discovery facilities on top of the architectural facilities provided by Prism-MW:

- **Topology-based routing.** Events are not tagged with the location of service provider; instead, they are routed based on the topology of the software architecture and the routing policies installed on the connectors. In turn, this makes a software system's architecture more flexible. For instance, clients are not aware of the location of the service provider. Even when the service provider changes location at runtime, *SDConnector*s route events to its new location via interacting with *Local/Global SDEngine*s.
- **Separation of concerns.** Application logic is completely separated from the service discovery protocol and its nuances. This is because unlike most state-of-the-art middleware solutions that require the application to perform service lookup and binding, in our approach it is entirely performed within meta-level connectors and components (i.e., *SDConnector, Local/Global SDEngine*). In fact, it is possible to change the service discovery implementation at runtime (i.e., by swapping the *SDEngine* and *SDConnector*), without modifying or bringing down the client components.
- **Efficiency.** By leveraging connectors, we are able to minimize the number of required distribution ports. This is because a connector in Prism-MW can perform selective routing of an event on a port's connections. Note that network sockets are extremely resource-expensive for embedded and resource constrained platforms. Therefore, it is often desirable to minimize the number of instantiated network sockets in a system. Our approach is unlike most state-of-the-art middleware solutions, where each client would have to connect directly to a service provider via a

separate distribution port.

- **Scalability.** Our approach performs a service lookup in a hierarchical manner, where a service lookup request is always first processed by a *Local SDEngine*. Only when the *Local SDEngine* does not have any information of the requested service, the *Global SDEngine* is queried. Therefore, our solution can be used effectively for pervasive applications running on a large number of platforms, such as MIDAS.

5. Replication and Failover Facilities

As mentioned earlier, one approach to providing failover support is through the replication of software components, and thus the services they provide. There are two types of replication strategies for supporting fault tolerance [13]:

1. **Active replication.** Each replica of a service provider (component) processes a service request event and returns a response to the client. Hence, duplicate detection is necessary for the client to receive only one response.
2. **Passive replication.** Only one of the replicas, selected as the primary replica, processes a request event and sends a response to the client. With *warm passive* replication, the remaining backup replicas are synchronized periodically with the primary replica's state. With *cold passive* replication, a backup replica is loaded into memory and its state initialized from a log only if the current primary replica fails.

Compared with passive replication, active replication has a faster recovery time because the failure of a single replica is simply masked by the presence of the other active replicas, while requiring more processing power as every replica processes each request event.

MIDAS has a stringent latency requirement of transmitting an event from a sensor to a hub and receiving an acknowledgement back in less than two seconds. Therefore,

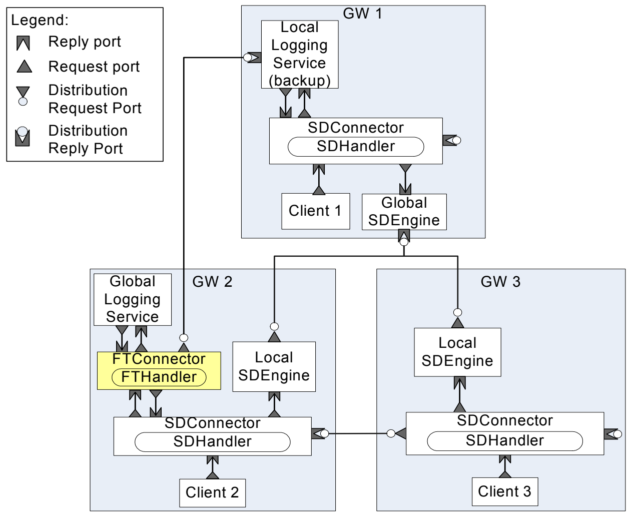

Figure 5. Fault tolerance support in the MIDAS system.

our replication strategy was based on the active replication style, as it results in a much shorter recovery time. Figure 5 shows our solution for replication synchronization and failover in the MIDAS system. We developed two implementations of *AbstractHander* (shown in Figure 2), called *BestEffortFTHandler*, and *RealTimeFTHandler*. Both *Handler*s not only deliver a service request event to the global service provider it was destined for, but also to the service provider's backup replicas. An *ExtensibleConnector* with an instance of *BestEffortFTHandler* or *RealTimeFTHandler* installed on it is called *BestEffortFTConnector* or *RealTimeFTConnector,* respectively. An *FTConnector* is placed between a global service provider (e.g., Global Logging Service component on Gateway 2) and its local *SDConnector* as shown in Figure 5. The two types of connector differ in their event delivery semantics as follows:

- ***BestEffortFTConnector.*** This connector sends the response received from a global service provider to the client only when the "ACK" messages are received from all the backups. If an "ACK" message is not received from a particular backup, the *BestEffortConnector* sends the request event to that backup again. If the connector still does not receive an "ACK" from the backup even after a pre-specified number of retransmits, it removes that backup from the list of backups, sends the response to the client, and informs the *Local/Global SDEngine*s of this backup failure via its local *SDConnector*, so that they can update their databases.
- ***RealTimeFTConnector.*** This connector does not wait for "ACK" messages from the backups, but sends the response received from a global service provider to the client immediately. Hence, a backup does not need to send an "ACK" message to the remote *RealTimeFTConnector* after processing an event. This connector is preferred in situations that a shorter response time to the client is more crucial than ensuring that all the active backups are synchronized with a global service provider.

To detect a host's failure, each *Local SDEngine* sends a heartbeat message periodically to the *Global SDEngine*. The *Global SDEngine* checks these heartbeat messages to determine a host's failure. When the *Global SDEngine* detects that a host has failed, for each global service provider on the failed host it promotes one of the backup replicas to take over, and informs all the *Local SDEngine*s of this promotion.

We have identified several advantages of providing replication and failover facilities on top of the architectural facilities:

- **Efficiency.** Typically, one of the main issues in supporting fault tolerance is to guarantee that all of the replicas of a service receive and process the request events in the same order. Otherwise, the replicas could end up in inconsistent states. For this, previous solutions [13,14] have relied on reliable totally-ordered multicast protocols such as Totem [12]. On the other hand, in our solution each request event is first routed to the *FTConnector* attached to a global service provider, and then forwarded to both a global service provider and its backups. Therefore, all the replicas receive the same sequence of request events. Compared with a traditional active replication style, our approach does not require the detection of duplicate responses from the replicas, since only the global service provider generates a response and the backups simply send an "ACK" message.

- **Extensibility.** Our approach can be easily extended to support other replication styles. For example, a global service provider can simply send its state periodically to all of its backups via the *FTConnector* for supporting a warm passive replication style.
- **Flexibility.** Addition and removal of replication support can be easily achieved at the architectural level via the addition or removal of an *FTConnector*. Similarly, if a new backup needs to be added to the system, it simply needs to be connected to the appropriate *FTConnector*.

6. Replication and Deployment Analysis

In order to effectively utilize the run-time fault tolerance facilities provided by Prism-MW, we have developed a complementary suite of modeling and analysis tools to guide replication and deployment decisions. As noted earlier, computational resources like memory, processing power and bandwidth are scarce in many pervasive systems, including MIDAS. Consequently, replication must be done selectively, targeting those services for which the greatest benefit will be attained. Furthermore, the system's deployment architecture (i.e., allocation of the system's software modules on its hardware hosts), may have a dramatic effect on the extent to which replication is possible and the degree of fault tolerance achieved. Enabling the software architect to make informed and judicious decisions in these areas is therefore critical to a comprehensive fault tolerance approach.

The decision of replication strategy and deployment architecture is complex in almost any industrial-scale system. Generally, the system components have varying degrees of reliability, and the replication of a highly reliably component provides less benefit than the replication of an unreliable component. Moreover, each component can collaborate with other components in multiple system use cases, which are of unequal importance to the user; that is, the services provided by some components are of higher criticality than others. Additionally, the system hardware can fail (e.g., due to the exhaustion of battery power), and the performance and availability of the wireless network is unpredictable. Finally, once a component has been identified as a candidate for replication, a node must be selected for the replica's deployment, taking into consideration the computational resources required and those available at each host.

To solve this multidimensional problem, we have developed an algorithm that determines a subset of software components that, once replicated, result in the greatest improvement in the system's fault tolerance. The replication decision algorithm iteratively identifies the next component for which the greatest benefit will be gained through replication by considering the utility of the combined services for which the component is required and the probability of the component's failure (the details of this algorithm are described in [16]). If replicating the component that achieves the highest value satisfies the computational resource constraints, it is suggested to the software architect as the best component to replicate next.

Our replication and deployment decision algorithms are implemented in DeSi [11], a visual environment that supports specification, analysis, and manipulation of a distributed software system's deployment architecture. DeSi provides an extensible meta-modeling language that can be used to specify arbitrary properties for software components and connectors (e.g., maximum memory usage, frequency and volume of exchanged data, mean time to failure), as well as hardware devices and their network links (e.g., CPU speed, available memory, network throughput). Users can leverage multiple views of a system model to explore and analyze its various properties. DeSi provides an API for accessing the system model, which is used to make analytical improvements to a given system's software architecture. For example, DeSi allows a software architect to improve the system's fault tolerance (via intelligent replication and deployment, as described above) within the context of a wider set of important QoS properties. This is vital, because while replication may improve fault tolerance, other QoS properties may degrade due to the overhead of executing additional components and keeping replicas synchronized.

The final element of our integrated fault tolerance strategy is the determination of component reliabilities, which impact heavily the replication and deployment algorithms, and ultimately the effectiveness of the runtime service discovery and failover support in Prism-MW. When it is not possible to measure component reliabilities empirically, these values may be estimated through simulation of architectural models in the XTEAM environment [3], a suite of architecture description language (ADL) extensions and model transformation engines targeted specifically for highly distributed, resource-constrained, and mobile computing environments. XTEAM implements (among other analyses) the component reliability estimation technique described in [17]. This approach relies on the definition of component failure types, the probabilities of those failures at different times during component execution, and the probability of and time required for failure recovery. Given an execution scenario (e.g., network loading conditions and service usage profiles), the XTEAM simulation provides a reliability value for each component, which may then be utilized by DeSi's replication decision algorithms. Together, DeSi and XTEAM have allowed us to perform rapid quantitative analyses of our work in a large number of scenarios.

7. Related Work

ArchJava [1] is an extension to Java that unifies software architecture with implementation, ensuring that the implementation conforms to architectural constraints. ArchJava currently has several limitations that would likely limit its applicability in the embedded and pervasive computing setting: communication between ArchJava components is achieved solely via method calls; and ArchJava is only applicable to applications running in a single address space. Aura [20] is an architectural style and supporting middleware for ubiquitous computing applications with a special focus on user mobility, context awareness, and context switching. Aura's performance and scalability have not been assessed in distributed, embedded and pervasive settings. Furthermore, neither one of the above two technologies provide fault tolerance support.

None of the middleware platforms discussed below provide native implementation facilities required for software architecture-based development in a manner that is suitable to embedded and pervasive systems. Orbix/E [4] is a lightweight CORBA ORB optimized for embedded applications, including its relatively small memory footprint. XMIDDLE [9] is a data-sharing middleware for mobile computing, which allows applications to share data that are encoded as XML with other hosts. Lime [6] is a Java-based middleware that provides a coordination layer that can be exploited for designing applications which exhibit either logical or physical mobility, or both. Finally, MobiPADS [2] is a reflective middleware that supports both active deployment of augmented services for mobile computing and dynamic adaptation for providing flexible configuration of resources.

8. Conclusions

The development of fault tolerant software systems in pervasive environments is a challenging task. The complexity of providing advanced fault tolerance facilities, such as component replication, replica synchronization, and failover, often results in rigid and unmanageable applications. In this paper, we have presented a novel approach to modeling and implementing fault tolerance facilities at the architectural level. This approach results in a separation of application logic from fault tolerance logic, which in turn aids the construction, analysis, and adaptation of the software system. We have leveraged these characteristics to quantitatively analyze various replication and deployment strategies. While our experience thus far has been very positive, a number of pertinent questions remain unexplored. Most importantly, one of the current shortcomings is the lack of fault tolerance support for middleware-level components (e.g., *Global SDEngine*), potentially resulting in single points of failure. In our future work, we plan to investigate these issues in more detail.

9. Acknowledgement

This work is sponsored in part by the National Science Foundation under Grant number ITR-0312780 and by Bosch Research and Technology Center.

10. References

[1] J. Aldrich, et. al. ArchJava: Connecting Software Architecture to Implementation. *ICSE 2002*, May 2002.

[2] A. Chan, et. al. MobiPADS: A Reflective Middleware for Context-Aware Mobile Computing. *IEEE Transactions on Software Engineering, Vol. 29, No.12*, December 2003.

[3] G. Edwards, et. al. Scenario-Driven Dynamic Analysis of Distributed Architectures. *FASE 2007*, March 2007.

[4] Orbix/E. http://www.iona.com/whitepapers/orbix-e-DS.pdf

[5] E. A. Lee. Embedded Software. *Advances in Computers (Marvin V. Zelkowitz, ed.), Academic Press*, London, 2002.

[6] LIME http://lime.sourceforge.net/

[7] S. Malek, et. al. A Style-Aware Architectural Middleware for Resource Constrained, Distributed Systems. *IEEE Transactions on Software Engineering.* March 2005.

[8] S. Malek, et. al. Reconceptualizing a Family of Heterogeneous Embedded Systems via Explicit Architectural Support. *IEEE Int'l Conf. on Software Engineering*, May 2007.

[9] C. Mascolo et. al. XMIDDLE: A Data-Sharing Middleware for Mobile Computing. *Personal and Wireless Communications*, Kluwer.

[10] N. Medvidovic, et. al. A Family of Software Architecture Implementation Frameworks. *WICSA 2002*, Aug. 2002.

[11] M. Mikic-Rakic, et. al. A Tailorable Environment for Assessing the Quality of Deployment Architectures in Highly Distributed Settings. *CD 2004*, Edinburgh, Scotland, May 2004.

[12] L. E. Moser, et. al. Totem: A fault-tolerant multicast group communication system. *Comms. of the ACM*. April 1996.

[13] P. Narasimhan, et. al. State Synchronization and Recovery for Strongly Consistent Replicated CORBA Objects. *DSN 2001*, July 2001.

[14] P. Narasimhan, et. al., Eternal-A Component-based Framework for Transparent Fault-Tolerant CORBA. *Software Practice and Experience*, Vol. 32, pp. 771-788, 2002.

[15] D.E. Perry, et. al. Foundations for the Study of Software Architectures. *Software Engineering Notes*, Oct. 1992.

[16] D. Popescu. Framework for Replica Selection in Fault-Tolerant Distributed Systems. Tech. Report USC-CSSE-2007-702, 2007.

[17] R. Roshandel, et. al. Estimating Software Component Reliability by Leveraging Architectural Models. *ICSE 2006,* May 2006.

[18] M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. *Prentice Hall*, 1996.

[19] M. Shaw, et. al. Abstractions for Software Architecture and Tools to Support Them. *IEEE Trans. on Software Engineering*, April 1995.

[20] J. P. Sousa, et. al. Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. *WICSA 2002*, Montreal, August 2002.