Toward Reliable Service Management in Message-Oriented Pervasive Systems

Chun-Feng Liao, Student Member, IEEE, Ya-Wen Jong, and Li-Chen Fu, Fellow, IEEE

Abstract—Reliability is one of the key challenges of pervasive systems. Numerous message-oriented architectures and service discovery protocols have been proposed to support service management in pervasive systems. Nevertheless, few researches have been done to improve the reliability of pervasive systems. This paper attempts to propose a reliable service management framework by formally defining a message-oriented service application model and protocols that facilitate autonomous composition, failure detection, and recovery of services. Proposed approaches are realized by constructing a developer’s toolkit that enables rapid-prototyping of services. We evaluate the proposed approach by first proving the reliability property and then conducting experiments on recovery rate and performance. The results show that the recovery rate can be greatly improved by the proposed approach. Furthermore, the services developed by using the proposed approach are capable of integrating heterogeneous software/hardware, and can be deployed in dissimilar sites with little efforts.

Index Terms—Services models, services architectures, services discovery architecture, service systems.

1 INTRODUCTION

Recent surge of research on systems in pervasive environments has given us new opportunities and challenges. In such systems, services are usually triggered by contexts which represent the situations of people, places, time, or devices [8]. As a result, the “event-triggered” nature of pervasive systems makes the Message-Oriented Pervasive System (MOPS) becomes a major point of research interest in recent years [21], [28], [34]. Messaging is an event-based mechanism that enables asynchronous communication and loosely coupled integration. Hohpe and Woolf [19] point out that when compared with other paradigms, messaging is considered more immediate than file transfer, better encapsulated than shared database, and more flexible than Remote Procedure Call (RPC)-style invocation.

The communications in MOPS is supported by the Message-Oriented Middleware (MOM). MOM creates a virtual “software bus” for integrating heterogeneous message publishers and subscribers, namely, the “nodes.” The logical pathways between nodes are called “topics.” Based on this architecture, the system provides services by chaining nodes and topics together. For instance, A, C, D, and F in Fig. 1 collectively provide an “adaptive air conditioner” service. In this service, A is a software adapter of wireless temperature sensors, C is a context interpreter that transforms raw data into high-level context data, D decides the commands to be taken by performing logical reasoning based on the context data, and F is responsible for controlling fans or air pumps based on messages coming from the COMMAND topic. The MOM has several advantages. First, they come up with simple and intuitive abstractions of node behaviors. More specifically, all node behaviors can be reduced to three types (to receive messages, to process messages, and to send messages). Second, nodes are easier to “mock” and test. In Fig. 1, node E can be tested separately without the presence of node A by using a “mock” node that feeds dummy sensor messages. In addition, MOM facilitates “separation of concerns,” that is, since each node is isolated by the topics, developers are capable of concentrating only on the logic of the node to be built without worrying about the interferences with other nodes. Finally, due to the loosely coupled nature of MOM, failures are isolated. In Fig. 1, if D fails, the failure is isolated by the topics, either C or F is aware of the failure.

Despite these advantages, there are still several challenges when designing MOPS: 1) Contrary to traditional enterprise systems, nodes in MOPS are highly dynamic since they can join, leave, or fail at anytime. Given that a service is a group of collaborating nodes, then composing a service means to discover, to select, and to activate appropriate nodes. However, MOM does not provide mechanisms to maintain and to keep track of the relationship between nodes and services. 2) As pointed out by Abowd [2] and Edwards and Grinter [10], “reliability” is one of the key challenges of pervasive systems. In this paper, we define the term “reliability” as the ability of a system to detect failed components and then to recover them from failure states eventually. MOM supports failure isolation, but neither failure detection nor recovery is addressed. 3) In typical pervasive environments such as Smart Homes, the people setting up and maintaining the systems are consumers with little technical knowledge. The solutions to such challenges have to make the system as autonomous as possible. The reliability challenge is a typical example: the system without autonomous failure detection and recovery capabilities may frustrate users from time to time, since they are hardly able to pinpoint the
work to convey high-level concepts, while recognizing that algebra (see Section 6.1). As a result, we use UML in this section. Attributes formally because of the preciseness of process and UML, and 3) it is easier to validate the desired system specified accurately, 2) a failure detection and recovery mechanism that is aware of service failures. Such mechanism describes how to detect failed nodes and then how to recover the failed service by either replacing the failed node by alternative node or restarting the failed node autonomously. In the following, we use the term “reliable service management” to refer to the two features mentioned above.

The objective of this research is therefore to design a reliable service management framework for MOPS under the challenges listed above. We begin by defining a service model, namely, Pervasive Service Application Model (PerSAM), which defines key abstractions, data structures, and a taxonomy of entities (see Fig. 2) in MOPS. The reason for defining a service model is that MOM only comes up with the “node” and “bus” abstractions, which are insufficient to facilitate reliable service management. Based on PerSAM, we present Pervasive Service Management Protocol (PSMP), an application layer reliable service management protocol for MOPS. It is important to point out that we focus on the application layer of the network stack, so that we do not take protocol issues at lower layers such as the reliability issues of UDP and IP multicast into account. In this paper, we describe PerSAM and PSMP by using Unified Modeling Language (UML) [3] and Communicating Sequential Processes (CSP) [18]. UML is useful in illustrating data structures (by using Class diagrams) and interacting flow (by using Sequence diagrams) visually. On the other hand, CSP is a member of the family of process algebra and is a widely used mathematical language for describing distributed and concurrent systems. The benefits of using the CSP are: 1) the models and protocols can be specified accurately, 2) it enhances the reproducibility of PerSAM/PSMP since CSP is more precise than pseudocode and UML, and 3) it is easier to validate the desired system attributes formally because of the preciseness of process algebra (see Section 6.1). As a result, we use UML in this work to convey high-level concepts, while recognizing that CSP is useful in increasing preciseness and reproducibility.

2 RELATED WORKS

Pervasive systems are difficult to design and maintain because of heterogeneous hardware, software, wiring protocols, and programming paradigms. Moreover, services in pervasive environments are highly dynamic. Therefore, many Service-Oriented platforms have been proposed to deal with the above problems since the rise of pervasive computing. They can be classified into two categories: process-centric and data-centric [39].

In the process-centric style, the distributed components collaborate by invoking sequences of remote calls. The flows of calls are controlled by programs, which search services in a centralized service registry and invoke services in a synchronized way. The Context Toolkits [30] and Service-Oriented Context-Aware Middleware (SOCAM) [14] fall into this category. The Context Toolkits attempts to address context issues in pervasive systems. In Context Toolkits, contexts are gathered by the context widgets as well as from context aggregators, and then they are turned into situational information by the context interpreter. The communicating messages between distributed entities are encoded by using XML in a proprietary way. SOCAM aims to provide an ontology-based rapid prototyping platform of services in pervasive environments. Holloway et al. [17] propose a pervasive platform based on the Representational State Transfer (REST) [11] architectural style. REST borrows the design philosophy from the World Wide Web by abstracting everything as resources labeled by Universal Resource Identifier (URI), which can be manipulated with standard HTTP methods such as GET or POST. RESTful platforms can be categorized to process-centric style. The major difference between RPC-based platforms and RESTful platforms is that instead of keeping track of states in remote procedures, the RESTful platforms transfer and update state information to and from remote resources. Due to the synchronous and centralized nature of process-centric style, the process-centric platforms suffer from many reliability issues. First, the distributed components of these platforms are usually tightly coupled: they are usually bound to a static network address. Hence, the components must start in a strict order. Moreover, if a service consists of a chained call sequences, all intermediates must be restarted when one of them fails. The failed services are hard to recover because all components must be shut down and then be restarted in a strict order. Finally, the distributed components communicate synchronously. Hence, service provider and service user both must be ready to communicate at the same time. The caller gets stuck when the callee fails or when it is highly loaded.

Recently, more loosely coupled and asynchronous data-centric styles such as Tuple-space (TS) [13] and MOM are proposed. TS is essentially an associative virtual shared
memory storing serialized objects. Distributed clients can read, write, or take serialized objects from TS server. Lifestreams [12] is a time-based architecture for managing electronic documents, which stores documents in the TS server. The EventHeap [21] is a TS-based platform that primarily focuses on interactive workspaces (smart office). The EventHeap includes a set of standard messaging formats and several applications composing of heterogeneous software and hardware. LIME [28] is a Java-based TS middleware that provides an abstraction layer supporting mobile computing environments. TS and MOM have similar advantages, that is, easy to integrate heterogeneous hardware/software and failure isolation. However, they are two different architectures from the technology’s point of view: TS is a way to access shared information across multiple concurrent clients, whereas MOM focuses on message delivery. More concretely, TS combines the concepts of centralized database and message delivery together. TS can simulate the event-driven feature of MOM, however, it tends to be less efficient as they are generally implemented using a remote accessible shared memory, which uses locks with read/writes to entries. Moreover, TS tends to store serialized objects, which is usually a penalty on performance and interoperability. MIRES [34] is one of the few pervasive systems based on MOM. It is designed mainly for Wireless Sensor Networks (WSN). However, MIRES focuses on the gathering of contexts, and doesn’t address reliability issues or how to compose services by grouping nodes of MOM.

As mentioned earlier, a service composition framework that is capable of discovering, selecting, and activating nodes is critical in pervasive systems. Many discovery protocols have been designed to support these features. Among them, Jini [20], Simple Service Discovery Protocol (SSDP), which is part of UPnP [38], SLP [32] and Salutation [31], have been discussed most widely. SSDP differs from other protocols in that it uses a more decentralized approach which is preferred in a pervasive environment. Current discovery protocols generally expect application to initiate recovery [7]. Hence, these discovery protocols provide few supports for failure detection, and none of them supports autonomous failure recovery. In this paper, we present a reliable service management framework, which carries out the recovery part and hence relieves considerable load from the application developers.

Context awareness allows a system to adapt to changing situations. Strang and Popien [36] report that five common approaches are used in context modeling: key-value, markup scheme, graphical, object-oriented, logic-based, and ontology-based. ContextServ [33] is a platform for context-aware Web Services based on ContextUML [29], a UML-based graphical context-modeling language. Strang and Popien also report that ontology-based approach is most expressive and fulfills most of requirements in their evaluation. The system proposed by [14], [15], [16], [27] fall into this category. Recently researchers are also interested in how to apply context-aware concepts to large-scale distributed systems [37]. In this work, we focus our attention on reliable service management. We use key-value context model in PerSAM/PSMP to gain higher focus. More discussions on related issues can be found in Section 7.

3 SERVICE MODEL

In this section, we will focus our attention on PerSAM. Several acronyms and notations are used throughout this paper to keep the presentation concise, which are summarized in Tables 1 and 2.

In PerSAM, the term “PerNode” refers to a basic logical software entity in MOPS. Note that we will use “PerNode” or “node” interchangeably in the following discussions. We divide PerNodes into two categories: the Manager Nodes and the Worker Nodes. Manager Node is designed for administrative purposes. Pervasive Service Manager (PSM) and Pervasive Host Manager (PHM) are both Manager Nodes, which are responsible for managing a Pervasive Service and a Pervasive Host (PH), respectively. On the other hand, Worker Nodes are basic useful functional units. We further classify Worker Nodes into three categories according to their behaviors: Sensor Nodes, Actuator Nodes, and Logic Nodes. Taking Fig. 1 as an example, A and B are Sensor Nodes, which are connected to gateways of sensors and the sensed data are sent to the SENSOR topic. Similarly, C and D are Logic Nodes that encapsulate logics of message processing. The computing device on which a PerNode are deployed is called a Pervasive Host. The procedures of installing a

---

### Table 1

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOM</td>
<td>Message-Oriented Middleware</td>
</tr>
<tr>
<td>MOPS</td>
<td>Message-Oriented Pervasive system</td>
</tr>
<tr>
<td>PerSAM</td>
<td>Pervasive Service Application Model</td>
</tr>
<tr>
<td>PSMP</td>
<td>Pervasive Service Management Protocol</td>
</tr>
<tr>
<td>PH</td>
<td>Pervasive Host</td>
</tr>
<tr>
<td>PHM</td>
<td>Pervasive Host Manager</td>
</tr>
<tr>
<td>PS</td>
<td>Pervasive Service</td>
</tr>
<tr>
<td>PSM</td>
<td>Pervasive Service Manager</td>
</tr>
<tr>
<td>PA/LA</td>
<td>Presence Announcement or Leave Announcement</td>
</tr>
</tbody>
</table>

### Table 2

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nt</td>
<td>An instance of node type</td>
</tr>
<tr>
<td>ps</td>
<td>An instance of Pervasive Service</td>
</tr>
<tr>
<td>ph</td>
<td>An instance of Pervasive Host</td>
</tr>
<tr>
<td>w</td>
<td>An instance of Worker Node</td>
</tr>
<tr>
<td>mp</td>
<td>An instance of PSM that manages ps</td>
</tr>
<tr>
<td>mph</td>
<td>An instance of PHM that manages ph</td>
</tr>
<tr>
<td>Wp</td>
<td>The set of Worker Nodes belonging to ps</td>
</tr>
<tr>
<td>Wh</td>
<td>The set of Worker Nodes belonging to ph</td>
</tr>
<tr>
<td>STp</td>
<td>The Service template of ps</td>
</tr>
<tr>
<td>MTp</td>
<td>The set of missed or failed node types that prevent ps from being alive</td>
</tr>
<tr>
<td>Tp</td>
<td>The set of timestamps that records the previous heartbeat time for each w ∈ Wp</td>
</tr>
<tr>
<td>m</td>
<td>A multicast channel</td>
</tr>
<tr>
<td>ti</td>
<td>An TCP-based unicast channel</td>
</tr>
<tr>
<td>un</td>
<td>An UDP-based unicast channel to node n</td>
</tr>
<tr>
<td>ssdp</td>
<td>An instance of SSDP message, x indicates the message type</td>
</tr>
<tr>
<td>Wnt</td>
<td>A list of candidate Worker Nodes with type nt</td>
</tr>
</tbody>
</table>
PerNode on a Pervasive Host are: 1) placing binaries of the node in a directory, and 2) registering its metadata so that it is manageable. After being installed, a node enters INSTALLED state. Next the node is loaded into memory, starting in DORMANT state. Note that although a DORMANT node does not perform any message-processing task, it still issues heartbeat and is “discoverable” by discovery protocols. A node goes into in ACTIVE state when it is activated. An activated node can receive, process, and send messages. Similarly, nodes can be removed from memory by a “shutdown” operation, or fall back to DORMANT state by a “rest” operation. The formal definitions of PerNode and Worker Node are as follows, which are depicted in Fig. 3:

**Definition 1 (PerNode).** A PerNode \( p \in P \) is an atomic stateful entity in MOPS, where \( P \) is the universe of PerNodes in the system, and \( state \in \{ \text{INSTALLED, DORMANT, ACTIVE} \} \) is an attribute of \( p \).

**Definition 2 (Worker Node).** A Worker Node \( w \in W \) is a PerNode that encapsulates a unit of application logic, where \( W \) is the universe of Worker Nodes in the system. In addition to the attributes inherited from PerNode, a Worker Node has three additional attributes: node type \( nt \in NT \), where NT is the universe of node types in the system, capabilities, and heartbeat period (hbp), which specify the functional category, capabilities of the Worker Node, and the heartbeat rate of the node, respectively.

### 3.1 The Pervasive Communities

A Pervasive Community is a logical organization of nodes. There are two kinds of Pervasive Communities: 1) the Pervasive Service (PS) consists of one or more nodes that collectively provide a service to user, and 2) the Pervasive Host refers to a group of nodes that colocate in the same computing device. Each community has a Manager Node that keeps track of its members. In other words, each PS has a PSM and each PH also has a PHM. A Worker Node can join several PSs at the same time. Considering an example depicted in Fig. 4, node A is a member of the “Adaptive Air Conditioner” PS (ps1) as well as the “Sensor Map” PS (ps2) at the same time.

Let us denote the PSM of a Pervasive Service \( ps \) as \( m_{ps} \) and the Worker Nodes that join \( ps \) as a set \( W_{ps} \in 2^W \), where \( 2^W \) is the power set of \( W \). Considering \( ps1 \) in Fig. 4, we have \( m_{ps1} = \{A, C, D, F\} \) and \( W_{ps1} = \{C\} \). The Service Template \( ST_{ps} \in 2^{NT} \) is predefined by service designers. Each \( ST_{ps} \) specifies required node types that comprise \( ps \). For example, in Fig. 4, \( ST_{ps1} = \{\text{Temperature Sensor, Context Interpreter, Indoor Temperature Control Logic, Air Conditioner}\} \). To compose a Pervasive Service \( ps \), \( m_{ps} \) first finds the best \( w \in W \) such that \( w:nt = nt \) for each \( nt \in ST_{ps} \). The definition of “the best” depends on a user-defined selecting function (see Table 3). The default selecting function is First Come, First Select (FCFS), but it can be substituted by more sophisticated ones.

Before \( ps \) is successfully composed, some Worker Nodes of required types \( nt \in ST_{ps} \) are still missing. Let us denote the set of node types of missing Worker Nodes as \( MT_{ps} \). Formally:

\[
MT_{ps} = \{nt | nt \in ST_{ps}, \exists w \in W_{ps} : w:nt = nt\}. \tag{1}
\]

In the previous example, assuming that \( W_{ps1} = \{C, D, F\} \), then \( MT_{ps1} = \{\text{Temperature Sensor}\} \), because the missing Worker Node A is of the type “Temperature Sensor.” In the beginning of service composition, since there are no Worker Nodes found, thus we have \( W_{ps1} = \emptyset \) and \( ST_{ps1} = \{\text{Temperature Sensor, Context Interpreter, Indoor Temperature Control Logic, Air Conditioner}\} \).

### Table 3

<table>
<thead>
<tr>
<th>Name</th>
<th>Input</th>
<th>Output</th>
<th>Definition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ServiceAlive</td>
<td>( \emptyset )</td>
<td>Boolean</td>
<td>see Def.3</td>
<td>Returns the liveness of a Pervasive Service</td>
</tr>
<tr>
<td>Refresh</td>
<td>( w \in W_{ps} )</td>
<td>Boolean</td>
<td>see Def.3</td>
<td>Update heartbeat timestamps of ( w ) with current time</td>
</tr>
<tr>
<td>Timeout</td>
<td>( w \in W_{ps} )</td>
<td>Boolean</td>
<td>( T^{ps}[w] := T^{ps}[w] + k )</td>
<td>Removes ( w ) from ( ps )</td>
</tr>
<tr>
<td>Remove</td>
<td>( w \in W_{ps} )</td>
<td>Boolean</td>
<td>( W^{ph} := W^{ph} - w )</td>
<td>Removes ( w ) from ( ps )</td>
</tr>
<tr>
<td>Add</td>
<td>( w \in W_{ps} )</td>
<td>Boolean</td>
<td>( W^{ph} := W^{ph} + w )</td>
<td>Add ( w ) to ( ps )</td>
</tr>
<tr>
<td>Select ( W_{nt} )</td>
<td>( w \in W_{nt} )</td>
<td>User defined</td>
<td>User defined</td>
<td>Returns a best node ( w ) from candidate list ( W_{nt} )</td>
</tr>
</tbody>
</table>

Fig. 3. The structure of a PerNode and a Worker Node.

Fig. 4. The Pervasive Service communities.
Definition 4 (Pervasive Service). A Pervasive Service ps is alive if and only if the following statement holds:

\[ MT_{ps} = \phi \land \forall w \in W_{ps}, \ w.\text{state} = \text{ACTIVE}. \]  

(2)

To detect possible failures of affiliating Worker Nodes, a PSM \((m^{ps})\) keeps track of timestamps of previous heartbeats \(t^{w}\) for each \(w \in W_{ps}\) in a vector of timestamps \(T_{ps}\), where \(t^{w} \in T_{ps}\) and \(|T_{ps}| = |W_{ps}|\). Based on the above discussions, we can formally define a Pervasive Service as follows:

Definition 4 (Pervasive Service). A Pervasive Service ps is a tuple:

\[ ps = (m^{ps}, ST_{ps}, MT_{ps}, W_{ps}, T_{ps}), \]  

(3)

where \(m^{ps}\) is PSM of ps, \(ST_{ps} \subseteq 2^{NT}\) is the Service Template, \(MT_{ps} \subseteq 2^{W}\) is set of missed types, \(W_{ps} \subseteq 2^{W}\) is the set of Worker Nodes join ps, and \(T_{ps}\) is a vector of timestamps that records the previous heartbeat time for each \(w \in W_{ps}\).

In similar way, we can now present the structure of a Pervasive Host. Each PH is composed of a set of Worker Nodes, whose life cycles are managed by a PHM. The PHM and its affiliating Worker Nodes locate in the same device. Therefore, a PHM is able to detect the node states, to load and to shutdown Worker Nodes. In Fig. 5, there are three PHs: A and B belong to \(ph_{1}\) since they are deployed in the same device; similarly, C, D, and F belong to \(ph_{2}\); the \(ph_{3}\) has single member E. Note that \(ph_{1}, ph_{2}\), and \(ph_{3}\) are managed by PHM 1, 2, and 3, respectively. The definition of a Pervasive Host is as follows:

Definition 5 (Pervasive Host). A Pervasive Host ph is a tuple:

\[ ph = (m^{ph}, W^{ph}), \]  

(4)

where \(m^{ph}\) is the PHM of ph, and \(W^{ph}\) is the set of Worker Nodes that currently locate on ph.

3.2 The Pervasive Managers

The responsibilities of a PSM are as follows:

1. To compose a PS according to Service Template \(ST_{ps}\). As mentioned in Section 3.1, when there are many qualified candidates, PSM first stores them in a candidate list denoted \(W_{st}^{ps}\) and then selects the best one by invoking the predefined selecting function.

2. PSM monitors all \(w \in W_{ps}\). Once PSM observes that a Worker Node does not heartbeat for longer than a predefined threshold, it emits a “suspect” message for that node.

3. PSM can add to or remove community members from PS.

4. PSM is responsible for keeping the PS alive.

In case PS is not alive, PSM attempts to recompose PS. As depicted in Fig. 6, there are six operations in PSM used to support the above-mentioned responsibilities. The input parameters, return values, definitions, and explanations of these operations are revealed in Table 3. Fuller discussion of how these operations work will be presented in the Section 4.

On the other hand, PHM is an agent that administrates nodes located in the same computing device. The tasks of PHM include monitoring and maintaining local Worker Nodes, loading local Worker Nodes from the file system into memory, and killing the failed Worker Nodes that do not emit heartbeat messages. It is important to point out that the task of discovering community members is faster and more reliable for PHM than for PSM, since no network-based communications are involved. The life cycles of Worker Nodes deployed in the same PH can be altered by the PHM locally. As indicated in Fig. 6, PHM contains two operations. The Shutdown operation removes a local Worker Node from memory. Similarly, the Load operation loads a local Worker Node into memory. The input parameters, return values, definitions, and explanations of these operations are shown in Table 4. Notice that the Install and Uninstall operations are invoked when a Worker Node installed to or uninstalled from a computing device.

4 Facilitating Reliable Service Management in MOPS

In this research, we realize PerSAM/PSMP by extending UPnP, a home networking protocol standard (ISO/IEC 29341) [38]. The reason for choosing UPnP is three fold: 1) it is one of the few dynamic service discovery protocols that do not need a dedicated and centralized service registry [7], which is more feasible for reliable service management. 2) UPnP is independent of platform and programming languages. 3) UPnP is a widely used and well-known standard. SSDP takes charge of service discovery in an UPnP network. By default, SSDP operates based on HTTPMU (HTTP over UDP multicast). HTTPMU uses IP multicast, which is supported by most network switching
equipments. IP multicast forwards packets to a group of interested receivers via a set of predefined virtual IP addresses. Therefore, SSDP does not need a centralized server since the multicast service is carried out by the underlying infrastructure. SSDP extends HTTP by two HTTP methods: NOTIFY and M-SEARCH. SSDP consists of three primitive actions: 1) ssdp:alive: announces the presence of a node, 2) ssdp:byebye: announces that a node has left the network, and 3) ssdp:discover: attempts to find a node that meets the type specified in ST (Search Target) header in a M-SEARCH message. The matched devices then reply by a HTTP Response message.

UPnP specifies a Device Architecture. An UPnP Device consists of a set of UPnP Services, and each UPnP Service comprises several UPnP Actions. A node that is capable of invoking UPnP Actions is called a Control Point, which can also be embedded in an UPnP Device. In this work, we implement PerNode based on UPnP Device Architecture (see Fig. 7). More precisely, each PerNode Device consists of an UPnP Service, the PerNode Life-Cycle Management Service, which manages PerNode life cycle by using three UPnP Actions (Activate, Rest, and Shutdown). Manager Nodes (PSM and PHM) are special types of PerNode Devices because they contain a Control Point. The reason for this design is that a Control Point is capable of invoking UPnP Actions of remote PerNodes to manage their life cycles. It is important to point out that despite the similarity in their names, the UPnP Services are different from Pervasive Services: an UPnP Service always embedded in an UPnP Device, while a Pervasive Service is a virtual community that consists of a group of nodes.

4.1 Presence Announcement, Leave Announcement, and Life-Cycle Management

Before we proceed, let us first take a look at some basic CSP syntax. CSP uses the form \( P \equiv e \rightarrow R \) to describe the behaviors of a Process \( P \), which first takes part in an event \( e \), and then behaves like process \( R \). Parameters can be passed to a process by enclosing with square brackets. For example, in \( P[x] \equiv f(x) \rightarrow R \), the \( x \) enclosed by square brackets is passed to the function \( f(x) \) in the right hand side. In CSP, \( c!m \) denotes an output event, in which a message \( m \) is emitted through network channel \( c \). In similar way, \( c?m \) denotes an input event, in which a message \( m \) is received through channel \( c \). A special process \( \text{SKIP} \) denotes a process that terminates without error. Table 5 summarized the CSP notations used throughout this paper.

Let us now return to PSMP. In PSMP, Presence Announcement (PA) and Leave Announcement (LA) can be formally described as follows:

In (5), PA sends an “ssdp:alive” to the multicast channel \( \hat{m} \) to announce the presence of the node \( p \), and then terminates. The same syntax applies to the definition of LA except that the message is “ssdp:byebye.” Based on (5) and (6), we can now define Life-Cycle Management (see Protocol 1). The Life-Cycle Management protocol (LM) enables Manager Nodes to change states of nodes remotely.

Protocol 1 (Life-Cycle Management). A Life-Cycle Management protocol changes state of PerNode according to incoming calls to UPnP Actions. A function \( \text{NewState} \) is used to decides new state based on the current state and the action being invoked. If a node is changed to INSTALLED state, it performs a leave announcement

\[
\text{LM}[p] \equiv \hat{m}\text{ssdpalive}[p] \rightarrow \text{SKIP}, \tag{5}
\]

\[
\text{LA}[p] \equiv \hat{m}\text{ssdpbyebye}[p] \rightarrow \text{SKIP}. \tag{6}
\]

In (5), PA sends an “ssdp:alive” to the multicast channel \( \hat{m} \) to announce the presence of the node \( p \), and then terminates. The same syntax applies to the definition of LA except that the message is “ssdp:byebye.” Based on (5) and (6), we can now define Life-Cycle Management (see Protocol 1). The Life-Cycle Management protocol (LM) enables Manager Nodes to change states of nodes remotely.

Protocol 1 (Life-Cycle Management). A Life-Cycle Management protocol changes state of PerNode according to incoming calls to UPnP Actions. A function \( \text{NewState} \) is used to decides new state based on the current state and the action being invoked. If a node is changed to INSTALLED state, it performs a leave announcement

\[
\text{LM}[p] \equiv \hat{m}\text{ssdpalive}[p] \rightarrow \text{SKIP}, \tag{5}
\]

\[
\text{LA}[p] \equiv \hat{m}\text{ssdpbyebye}[p] \rightarrow \text{SKIP}. \tag{6}
\]

Based on (5) and (6), we can now define Life-Cycle Management (see Protocol 1). The Life-Cycle Management protocol (LM) enables Manager Nodes to change states of nodes remotely.

Protocol 1 (Life-Cycle Management). A Life-Cycle Management protocol changes state of PerNode according to incoming calls to UPnP Actions. A function \( \text{NewState} \) is used to decides new state based on the current state and the action being invoked. If a node is changed to INSTALLED state, it performs a leave announcement

\[
\text{LM}[p] \equiv \hat{m}\text{ssdpalive}[p] \rightarrow \text{SKIP}, \tag{5}
\]

\[
\text{LA}[p] \equiv \hat{m}\text{ssdpbyebye}[p] \rightarrow \text{SKIP}. \tag{6}
\]
4.2 Service Composition

The purpose of service composition is first to find appropriate Worker Nodes for a PS, and then to ensure that the chosen nodes are in ACTIVE states persistently. Fig. 8 illustrates the interactions between these nodes when performing service composition. Whenever the service is not alive (see Definition 3), the PSM issues a “psmp:discover” to find PS members (Fig. 8, step 1) to initiate a service composition. Finding qualified Worker Nodes by simply issuing an “ssdp:discover” causes low degree of support and low composition sustainability [23]. The reason is that “ssdp:discover” only discovers nodes that are already loaded into memory (i.e., DORMANT state). To put it another way, the nodes that are in INSTALLED state do not respond to PSM. A typical case is that for a newly booted system, nearly all nodes are in INSTALLED states. Hence, few services can be successfully composed in this circumstance. PSMP deals with the problem mentioned above by defining a new “psmp:discover” action. This action is issued by PSM to perform “eager loading” of nodes. In simple terms, when PHM receives “psmp:discover,” it loads all qualified local nodes that are in INSTALLED state. Once the qualified nodes become DORMANT, they will send presence announcements so that PSM will be able to discover them (Fig. 8, step 3-5). The “psmp:discover” can discover nodes that are not loaded, therefore it solves the problem mentioned above. The behavior of PSM in service composition is formally defined below.

Protocol 2 (PSM Service Composition). A PSM Service Composition \((SC_{PSM})\) is initiated whenever the Pervasive Service is not alive. For each node type in \(MT_{PS}\), PSM issues a discovery message \((m\text{-search})\) to the multicast channel \((\hat{m})\) and then performs PSM Service Selection \((SS_{PSM}, \text{see Protocol 5})\), formally

\[
SC_{PSM}[ps] \triangleq \begin{cases} \hat{m}(&\neg Service.Alive()) \\ \prod_{nt \in MT_{PS}} \hat{m}_{ssdp}^{m\text{-search}}[nt] \to SS_{PSM}[ps] \\ \text{else } SC_{PSM}[ps]. \end{cases} \tag{9}
\]

In (9), the \(\prod\) operator is a shorthand for iteration. For instance, \(\prod_{nt \in MT_{PS}} P\) means that a process \(P\) executes one time for each \(nt \in MT_{PS}\). Upon receiving a discovery message, a qualified Worker Node responds with a message in which describes its accessing information (Fig. 8, step 1.1 and step 3). Protocol 3 describes the behavior for Worker Node in service composition. In (10), \(\hat{u}_{psm}\) represents a UDP unicast channel corresponding to the searching PSM.

Protocol 3 (Worker Node Service Composition). A Worker Node Service Composition \((SC_W)\) examines incoming discovery messages. If there is a match in its node type, then it sends an response message \((ssdp\text{-resp})\) indicating its accessing information via the UDP unicast channel \((i_{psm})\) corresponding to the source of the discovery message

\[
SC_W[ps] \triangleq \begin{cases} \hat{m}(&\text{?ssdp}^{m\text{-search}}[psm, nt] \to \begin{cases} \text{if} (w.n = nt) \text{then } \hat{u}_{psm}^{ssdp}\text{-resp} \to SC_W[w] \\ \text{else } SC_W[w]. \end{cases} \end{cases} \tag{10}
\]

Meanwhile, PHM is responsible for discovering INSTALLED nodes. Upon receiving a “psmp:discover,” PHM compares the node type against local INSTALLED nodes (Fig. 8, step 3). If there is a match, then PHM loads the node, causing it enters the DORMANT state (Fig. 8, step 4).

Protocol 4 (PHM Service Composition). A PHM Service Composition \((SC_{PHM})\) examines incoming discovery messages. According to the specified node type, a PHM iteratively match against local nodes and then load the matched node into memory (i.e., DORMANT state). It also emits a response message for the matched node

\[
SC_{PHM}[ph] \triangleq \begin{cases} \hat{m}(&?ssdp^{m\text{-search}}[psm, nt] \to \begin{cases} \prod_{w \epsilon ph} \text{LS}[w, psm, nt, ph] \\ \begin{cases} \text{if} (w.n = nt) \text{then } \hat{u}_{psm}^{ssdp}\text{-resp} \to SC_W[w] \\ \text{else } SC_W[w]. \end{cases} \end{cases} \end{cases} \tag{11}
\]

In (12), Load is an operation of PHM (see Table 4). PSM selects and activates Worker Nodes for a PS. The protocol for PSM Service Selection and Activation is shown below:

Protocol 5 (PSM Service Selection and Activation). A PSM Service Selection protocol \((SS_{PSM})\) examines responses from Worker Nodes. For each response, PSM add the node into a list of candidates for a specific node type \((W^c)\). After \(MT_{PS}\) being empty, PSM selects the best ones for each node type according to a user-defined selecting function. After that, it executes PSM Service Activation \((SA_{PSM})\)
\[ SS_{PSM}[ps] \triangleq \hat{u}\text{ssdp}^{\text{research}}[w] \]
\[ \text{if}(MT^{ps} = \phi) \]
\[ \text{then} \bigwedge_{nt \in ST^{ps}} \left[ \text{Add}(\text{Select}(W_{nt})) \rightarrow S_{PSM}^{ps}[\text{Select}(W_{nt}), ps] \right] \]
\[ \text{if}(w.nt \in MT^{ps}) \]
\[ \text{then} \ MT := MT - w.nt \]
\[ \text{else} \]
\[ \text{else} \ S_{PSM}^{ps}[ps] \]
\[ S_{PSM}^{ps}[w, ps] \triangleq \hat{v}\text{call}[\text{activate}] \rightarrow W_{nt}^* := \phi \rightarrow S_{PSM}^{ps}[ps]. \] (13)

In \( S_{PSM}^{ps} \), PSM invokes an UPnP Action (activate) of the selected Worker Node, and then reset \( W_{nt}^* \). After all Worker Nodes in a PS are activated, the PS becomes alive.

### 4.3 Failure Detection and Recovery

To keep a PS alive, all affiliating Worker Nodes must be in \textit{ACTIVE} state lastingly. If one of them fails, then the PS becomes unavailable. Therefore, PSM has to be aware of the failures of Worker Nodes first (Failure Detection) and then resumes or substitutes the failed ones (Service Recovery). As noted in Section 1, we use the term “reliable service management” to refer to the mechanism that enables a system to detect failures and to recover from failures autonomously. Having clarified the semantics of reliability, we are now able to formally define a Reliable Pervasive Service.

**Definition 6 (Reliable Pervasive Service).** A Pervasive Service \( ps \) is reliable if and only if the following statement holds:
\[ \diamond \text{Fail}(w^f) \Rightarrow \neg \diamond \text{ServiceAlive}() \land \diamond \text{ServiceAlive}(), \] (15)
where \( w^f \in W^{ps} \) is a failed Worker Node, and \( \text{Fail}(w^f) \) represents the fact that \( w^f \) fails.

The symbol \( \diamond \) is used to denote “eventually” and the symbol \( \neg \) to denote “always” in the logic statements. These symbols are borrowed from Temporal Logic [24]. Note that \( \neg \text{ServiceAlive}() \) happens before \( \diamond \text{ServiceAlive}() \); therefore, their conjunction is not necessarily false.

In this section, we present protocols that ensure the reliability of PS. These protocols are designed based on the following assumptions.

1. **Eventually Correct Local Failure Detector (A1).** A Worker Node stops performing heartbeat eventually if it fails. This assumption is made in theory for gaining rigor of the result to be obtained. In most real cases, this assumption can be fulfilled by implementation techniques which are platform dependent. For example, in embedded Linux platforms, the local failure detectors can be implemented by means of either built-in system monitoring hardware or “watchdog” services included in standard kernel packages [40].

2. **Perfect-Link Assumption (A2).** Since we concentrate on application layer in this research work, a Perfect-Link model is assumed, in which all messages are guaranteed to be successfully delivered. In addition, a message does not appear in the network unless a node sends one. In network layer, this assumption can be ensured by using reliable multicast protocols such as RMP [41] or SRM [42].

3. **Persistent Manager Nodes Assumption (A3).** Manager nodes will not experience failure, of which such assumption is made since we don’t consider the reliability issues of Manager Node for the time being. We are recently designing consensus-based protocols that make the failures of Manager Nodes detectable and recoverable without centralized coordinators [22]. When failure detection protocols for Manager Nodes are absent, one simple yet effective solution is to use the watchdog services provided by the underlying platform to detect and to recover the failed Manager Nodes.

4. **Composable Service Assumption (A4).** All services are composable. In other words, for each PS, for all, there is at least one node of each type exists in the system. If this assumption is not held, then it is impossible to recover the PS.

The PSM failure detection is shown in Fig. 9. In the following, we formally define the behaviors of PSM, PHM, and Worker Node depicted in Fig. 9.

**Protocol 6 (Worker Node Heartbeat).** A Worker Node performs heartbeat by emitting PA periodically. The Worker Node attribute \( hbp \) is a predefined interval between each heartbeat.

\[ HB_{W}[w] \triangleq \text{sleep}(w.hbp) \rightarrow PA[w]; HB_{W}[w]. \] (16)

Protocol 7 reveals how PSM emits suspecting message. There are two processes running in parallel, one for refreshing \( T^{ps} \) and the other for timeout eviction (\( EV_{PSM} \)). In (17), \( \| \) is used to combine two parallel processes. In Protocol 7, Refresh, Timeout, and Remove are operations of PSM (see Table 2).

**Protocol 7 (PSM Node Suspecting).** PSM Node Suspecting protocol checks if there is an affiliated node stops performing heartbeat. If PSM does not receive any heartbeat for more than a predefined interval, it sends a suspecting message indicating a possible node failure.

\[ NSU_{PSM}[ps] \triangleq \left[ \begin{array}{c} \text{\hat{m}\text{ssdp}^{alive}}[w] \rightarrow \text{Refresh}(w) \rightarrow \text{NSU}_{PSM}[ps] \end{array} \right] \]
\[ \| \text{EV}_{PSM}[ps], \] (17)
\[ EV_{PSM}[ps] \triangleq \left[ \begin{array}{c} \text{\|} \ \text{\|} \ \text{\|} \end{array} \right] \left[ \begin{array}{c} \text{if } \text{Timeout}(w) \text{ then Remove}(w) \rightarrow \text{\hat{m}\text{psmp}}^{suspect}[w] \end{array} \right] \rightarrow \text{EV}_{PSM}[ps]. \] (18)

After a node is suspected, PHM stops the node and then sends a leave announcement on behalf of it (Fig. 9, step 5). These operations are described as follows:

**Protocol 8 (PHM Shutdown Suspect).** PHM Shutdown Suspect protocol stops the suspected nodes according to the
PSMP is a composite protocol that describes interactions between PSM, PHM, and Worker Nodes to realize reliable Pervasive Services.

\[ PSM[p_s] \triangleq PA[p_s]; (SC_{PSM}[p_s])||NSU_{PSM}[p_s]. \]  
\[ PHM[p_h] \triangleq PA[p_h]; (SC_{PHM}[p_h])||SSU_{PHM}[p_h]. \]
\[ W[w] \triangleq PA[w]; (SC_W[w])||HB[w]||LM[w]. \]

### 4.4 Running Scenario

This section summarized the service models and protocols mentioned above by providing a running scenario that illustrates service composition, failure detection, and recovery procedures of PSMP.

Let us consider a four-node Pervasive Service \( ps_1 \) depicted in Fig. 4, where \( ST^{ps_1} = \{ \text{Temperature Sensor, Context Interpreter, Indoor Temperature Control Logic, Air Conditioner} \} \). Initially, \( W^{ps_1} = \phi \) and \( MT^{ps_1} = ST^{ps_1} \). From Definition 3, \( ServiceAlive() = false \), so that \( SC_{psm} \) is triggered. For example, the statement \( m!ssdp\text{research}[\text{TemperatureSensor}] \) indicates that an m-search message is emitted to search nodes of the type “Temperature Sensor” (refer to Fig. 8, Step 1). After that, \( SS_{psm} \) is triggered, which listens for the responses from qualified nodes. Let us assume that node \( A \) is a “Temperature Sensor” node and that it responds to the discovery request (Fig. 8, Step 2). Given FCFS selection policy is used, from (10) and (13), node \( A \) is selected (Fig. 8, Step 6). Hence, \( MT^{ps_1} = \{ \text{Context Interpreter, Indoor Temperature Control Logic, Air Conditioner} \} \) and \( W^{ps_1} = \{ A \} \). In similar way, \( m^{ps_1} \) is able to discover node \( C, D, F \) with the node type Context Interpreter, Indoor Temperature Control Logic, and Air Conditioner, respectively, and causes \( MT^{ps_1} = \phi; W^{ps_1} = \{ A, C, D, F \} \). Finally, according to (13), \( SA_{psm} \) is triggered (Fig. 8, Step 7-8), and thus \( \forall w \in W^{ps_1}, w.\text{state}=ACTIVE \). Now that \( ServiceAlive() = true \) and that \( ps_1 \) is successfully composed.

After \( ps_1 \) is composed, its affiliating Worker Nodes perform heartbeats (16) periodically (see Fig. 9, step 1 and step 2). In (3), there is a set \( T^w \) used to store the previous timestamps of heartbeat for each node. If node \( A \) fails, then node \( A \) eventually stops heartbeat, causing (17) to emit a suspect message against \( A \) and \( MT^{ps_1} = \{ A \} \) (Fig. 9, step 3).

According to (19), upon receiving suspect message, \( phm \) removes \( A \) from memory (Fig. 9, step 4). Finally, \( ServiceAlive() = false \) and then \( SC_{psm} \) is triggered again in order to recompose \( ps_1 \). Given that node \( B \) is a “Temperature Sensor” node and that it responds to the discovery in the first place; then node \( B \) is activated and \( ps_1 \) is recovered.

### 5 Implementation

We have realized PerSAM mainly by using Java programming language on JDK 6. Some nodes are implemented by using C# and others using C++. We use ActiveMQ [1], an open-source MOM as the message exchanging backbone. It uses a platform independent messaging protocol, and thus a Pervasive Service can be composed of Worker Nodes that are implemented by different programming languages. This ability is important in developing pervasive applications. There are many CSP-based libraries or toolkits available such as JCSP [6] and PAT [35]. These tools are valuable in the design and validation phase of a protocol. However, they lack the abilities of performing low level control on packets (such as SSDP packet modification). In addition, these implementations are usually alternatives to the original thread model; it can be dangerous to mix them with native thread libraries. Consequently, PSMP is currently implemented by using UPnP libraries, which uses native network and thread libraries. Specifically, we use Intel UPnP SDK for C# and C++ based PerNodes, and Java-based PerNodes are developed based on Cyberlink UPnP.

To facilitate rapid prototyping, we have built a Java-based Object-Oriented application framework that provides design time supports with a set of reusable libraries, interfaces, and default implementations. One of its distinguishing features is that it supports attribute-based programming [4]; therefore the code becomes intuitive and more comprehensible. In addition, we have implemented drag-and-drop code generation services by a set of “Interactive Wizards,” which are realized as plug-in modules of the Eclipse IDE [9].
6 Evaluation

In this section, we evaluate the proposed approach of reliable service management in MOPS, namely, PerSAM/PSMP. The following sections explain the evaluations with respect to reliability, recovery capability, performance, feasibility, and cost.

6.1 Proofs of the Reliability Property

In this section, we prove the reliability property of PerSAM/PSMP. The objective is to validate that (15) holds for, which is stated as follows:

Theorem 1. PSMP-based Pervasive Services are reliable.

Before presenting the proof, we define an auxiliary function \( \eta : E \rightarrow \text{Boolean} \) that maps an CSP event to a logical assertion, where \( E \) is a set of CSP events. For example, \( \diamond \eta(e) \) represents the fact that an event \( e \) eventually happens.

Lemma 1. A failed Worker Node does not send any PA after the failure occurs eventually.

Proof. From (A2), all messages are guaranteed to be delivered to their sinks; and a message does not appear in the network unless a node sends one, hence the following statements hold

\[
\eta(e!x) \Rightarrow \diamond \eta(e?x) \tag{23}
\]

and

\[
\neg \eta(e!x) \Rightarrow \neg \neg \eta(e?x). \tag{24}
\]

Assume there is a Worker Node \( w^f \) fails, according to (A1), (22) and (16) must stop working. From (5), no PA will be sent from \( w^f \) after this failure. As a result, we have

\[
\eta(\text{Fail}(w^f)) \Rightarrow \neg \eta(\text{msg}_\text{ssdpAlive}[w^f]). \tag{25}
\]

\hfill \Box

Lemma 2. After \( w^f \) fails, it will eventually be removed from its affiliating Pervasive Service \( ps \) and a suspecting message with respect to \( w^f \) will be sent.

Proof. From (24) and from (25) the following statement holds

\[
\neg \eta(\text{msg}_\text{ssdpAlive}[w^f]). \tag{26}
\]

Thus, from (17) and from (26), \( \text{Refresh}(w^f) \) never executes after the failure, which causes \( \text{Timeout}(w^f) \) returns true. Finally, from (18), \( \text{Remove}(w^f) \) and \( \text{msg}_\text{psmpSuspend}[w^f] \) will occur. To sum up, from (26)

\[
\neg \eta(\text{Refresh}(w^f)) \Rightarrow \text{Timeout}(w^f) \Rightarrow \neg \eta(\text{Remove}(w^f)) \land \neg \eta(\text{msg}_\text{psmpSuspend}[w^f]). \tag{27}
\]

\hfill \Box

Lemma 3. Eventually, the failure of \( w^f \) will be detected, causing the PS of \( w^f \) becomes unavailable.

Proof. By definition, \( \text{Remove}(w^f) \) causes \( W^{ps} := W^{ps} - w^f \), thus \( w^f \notin W^{ps} \). In addition, since \( w^f . nt \in ST^{ps} \) since originally \( w^f \) is a community member of \( PS \). By combining (1) and (9), we have

\[
\diamond \eta(\text{Remove}(w^f)) \Rightarrow \diamond w^f . nt \in MT^{ps} \Rightarrow \diamond MT^{ps} \neq \phi \Rightarrow \neg \text{ServiceAlive}(). \tag{28}
\]

By combining (25), (26), (27), and (28), a failure is eventually detected, namely,

\[
\eta(\text{Fail}(w^f)) \Rightarrow \neg \text{ServiceAlive}(). \tag{29}
\]

\hfill \Box

Lemma 4. Eventually, PSM finds alternative nodes by performing node discovery for the type of \( w^f \).

Proof. The results is readily obtained from (9) and (28):

\[
\neg \text{ServiceAlive}() \land \diamond w^f . nt \in MT^{ps} \Rightarrow \diamond \eta(\text{msg}_\text{ssdpSearch}[w^f . nt]). \tag{30}
\]

\hfill \Box

Lemma 5. Eventually, there must be some alternative nodes of the type \( w^f . nt \) that respond to the node discovery.

Proof. It is easy to observe from (23) and (30) that the following statement holds:

\[
\diamond \eta(\text{msg}_\text{ssdpSearch}[w^f . nt]). \tag{31}
\]

In addition, (A5) states that \( \forall nt \in ST^{ps}, \exists w : w . nt = nt \). Since \( MT^{ps} \subseteq ST^{ps} \), we obtain the following statement:

\[
\forall nt \in MT^{ps}, \exists w : w . nt = nt. \tag{32}
\]

Since \( w^f . nt \in MT^{ps} \), from (32), there must be at least one alternative node \( w^f \) such that \( w^f . nt = w^f . nt \), in other words, \( \exists w^f . nt = w^f . nt \). By combining (10), (11), (12), and (32), we have

\[
\forall nt \in MT^{ps}, \exists w^f : w^f . nt = nt \land \diamond \eta(\text{msg}_\text{ssdpResp}[w^f]). \tag{33}
\]

\hfill \Box

From (13), (23), and (33), and the definition of \( \text{Add} \) (see Table 4),

\[
\diamond \eta(\text{msg}_\text{ssdpResp}[w^f]) \Rightarrow \diamond \eta(MT^{ps} := MT^{ps} - w^f . nt) \Rightarrow \diamond MT^{ps} = \phi \Rightarrow \diamond \eta(W^{ps} := W^{ps} \cup w^f) \Rightarrow \diamond w^f . \text{state} = \text{ACTIVE}. \tag{34}
\]

From (2) and (33), \( \text{ServiceAlive}() \) holds. By combining (29), (30), (31), (33), and (34), we have

\[
\eta(\text{Fail}(w^f)) \Rightarrow \neg \text{ServiceAlive}() \land \text{ServiceAlive}(). \tag{35}
\]

Consequently, we can validate that a PSMP-based Pervasive Service is reliable.

6.2 Recovery Rate

We employed simulations to study the PS recovery rates of SSDP and PSMP under different failure rates of Worker Nodes. The simulation environment consisted of 100 Worker Nodes, 10 PSs, and 3 PHs. Each PS consisted of three Worker Nodes with different node types. Initially, node types were evenly distributed to the Service Templates and to the Worker Nodes. After that, all PSs were composed and made available. In each experiment, Worker Nodes were
randomly terminated (crashed) according to a failure rate. For the experiments, hardware failure rate was 40 percent. That is to say, 40 percent of the failures were hardware or network interface failures, and thus were unrecoverable by software-based mechanisms. The experiments were performed 1,000 rounds under each failure rate. The average percentages of recoverable PSs were then recorded. Figs. 10 and 11 show the influences of Worker Node failure rate on the PS recovery rates when the number of node types is 25 and 50, respectively. The recovery rates of SSDP dropped rapidly when the failure rates of Worker Nodes increased. On the contrary, the recovery rates of PSMP decreased much slower. The results suggest that PSMP is superior to SSDP in the recovery capability. It is noteworthy that with PSMP, significant portion of PSs were able to be recovered even when the failure rates arrived 100 percent. This is because SSDP only discovers nodes that are already loaded into memory (i.e., in DORMANT or ACTIVE state). As opposed to SSDP, PSMP is capable of discovering the nodes that are in INSTALLED states via PHMs. The results also indicate that the number of node types has great impact on the recovery rate since it affects the number of alternative node for each node type.

### 6.3 Performance

We conducted experiments in realistic home network to study the performance of PSMP. Experiments consisted of two parts: 1) in the first experiment, the objective is to measure the turnaround time of service composition under different service lengths, and 2) experiments on measuring the recovery time helped us to investigate the trade-offs between eviction threshold $k$ (see Table 2) and the recovery time.

All nodes were deployed on Knopflerfish 2.0.1 OSGi servers, which were evenly distributed over three P4 1 GHz mini-PCs in the same LAN with 1G bytes memory. The environment consisted of one PS and three PHs. In each PH we deploy 50 Worker Nodes, whose node types were configured so that all PS can be composed successfully. We obtained the turnaround time of service composition by measuring the time from the PSM was started to the time when all required Worker Nodes were activated. After that, we increased the size of PS and reperformed the tests. The experiments were performed 100 times under each configuration of service length and then the average percentages of recoverable PSs were reported. The results are shown in Fig. 12. The turnaround time of service composition increased linearly when the number of Worker Nodes in PS increased. In our experiences, most real-world PSs consist of less than 10 nodes. Hence, we can observe in Fig. 12 that most real-world PSs require less than 1.2 seconds before it is available. It is also interesting to note that due to each node were executed in parallel, the average turnaround time for each node in a PS decreased as the service length increased. The second experiment was performed in similar ways with the previous experiment except that after the PS was composed, one Worker Node was randomly terminated. After that, we recorded the time from the Worker Node failed to the time when the PS is resumed. After that, we increased the size of PS and reperformed the tests. Fig. 13 indicates the performance of failure detection and recovery. The results show that the eviction threshold $k$ has great impact on the recovery time. This is because $k$ determines the upper bound of failure detection time. If $k$ is set as 500 ms, the total service unavailable time is less than 2 seconds.

### 6.4 Feasibility and Costs

We study the feasibility of PerSAM/PSMP by developing several services, which are deployed in two dissimilar sites. The details of these services can be found in [25]. These sites are different in size, partition, appliances, and furnishing. Due to their dissimilarities, developers have to modify configuration files for each site in order to deploy services. However, they do not change the source code. Also notice that the nodes can be implemented in different programming languages. For instance, the real-time image-processing components are better implemented with C or C++ while server-side components are usually implemented with Java language. This interoperability makes PerSAM highly extensible.

Nevertheless, the proposed approach imposes some costs and limitations. First of all, the hierarchical architecture
can be a cost because of the inclusion of Manager Nodes. The reason for this design is because decentralized failure detection and recovery such as consensus protocols are usually not efficient and are less scalable. We suggest a hybrid architecture that employs a centralized approach for Worker Node and a consensus-based approach for Manager Nodes. It is more cost effective when the number of Worker Nodes is much larger than that of Manager Nodes. Second, we design PSMP by extending SSDP. Obviously, there are interoperability costs imposed by this approach. However, PSMP does not interfere with the traditional UPnP Devices. This is because the use of “psmp:” headers. According to the UPnP specification, traditional UPnP Devices do not process the headers other than “ssdp:”.

7 CONCLUSION

Reliability is one of the most important aspects of a pervasive system especially when it is deployed in a smart space such as a smart home. In this paper, we present reliable service management mechanisms, namely, Per-SAM/PSMP, to facilitate autonomous service composition, failure detection, and recovery in MOPS. The proposed approaches are described formally by using CSP. Because of these formulations, we have proved the reliability of PerSAM/PSMP. In [26], we also observe that CSP-based formulation also makes the analysis of communication complexity easier. In addition, we have realized PerSAM/PSMP by constructing a developers’ toolkit, which enables rapid developments of services in MOPS. The toolkit consists of a reusable object-oriented application framework as well as toolkits that enable wizard-based/drag-and-drop styles of code generation. The experimental studies show that PSMP has much higher recovery rate than SSDP and is able to recover significant portions of PSs even when the failure rate reaches 100 percent. The performance evaluations show that for real-world PSs, service composition and failure recovery can be performed within 2 and 0.5 seconds, respectively. Finally, we show the feasibility of the proposed approaches by developing several PSs based on the above-mentioned toolkits.

This work concentrates on reliable service management, and has not touched the context-aware issues in pervasive systems. Apparently in the future many researches have to be done to model, store, or manage contexts in PerSAM/PSMP. It is important to point out that although we use key-value-based modeling techniques in this work; PerSAM/PSMP is designed independent of context model techniques. We are investigating context and preference aware service composition based on PerSAM/PSMP [5]. Another possible future work is to take user preferences and policies into account. Complexity arises when multiple parties submit conflicting preferences simultaneously. For example, the preferences of energy saving policies and the user preferences are very likely to be conflicting. PerSAM/PSMP can be extended to deal with these conflicts. In this paper, we assume Manager Nodes never fail. Further research is obviously required to refine the consensus recovery methods for PSM and PHM. Besides, in a heavy-loaded network, UDP is very likely to lose packets thus causing SSDP to become unstable. Some improvements on efficiency of PSMP can be found at [26].

ACKNOWLEDGEMENTS

This research is supported by the National Science Council of the Republic of China under Grant NSC 97-3114-E-002-002.

REFERENCES


Chun-Feng Liao received the BS and MS degrees in computer science from National Chengchi University in 1998 and 2004, respectively. He is currently working toward the PhD degree in the Department of Computer Science and Information Engineering at National Taiwan University. His research interests are context-aware and service-oriented architecture for smart living spaces. He is a student member of the IEEE and the IEEE Computer Society.

Ya-Wen Jong received the BS degree in computer science and information engineering from National Central University in 2007. She is currently a graduate student in the Department of Computer Science and Information Engineering at National Taiwan University. Her research interests include dynamic service management and context-aware and smart living spaces. He is a student member of the IEEE Robotics and Automation Society and the IEEE Automatic Control Society, and became a fellow of the IEEE in 2004.

Li-Chen Fu received the BS degree from National Taiwan University in 1981 and the MS and PhD degrees from the University of California, Berkeley, in 1985 and 1987, respectively. Since 1987, he has been on the faculties of both the Department of Electrical Engineering and the Department of Computer Science and Information Engineering of National Taiwan University. His research interests include robotics, FMS scheduling, shop floor control, smart home, visual detection and tracking, and control theory and applications. He is a senior member of the IEEE Robotics and Automation Society and the IEEE Automatic Control Society.