ADDING META-ARCHITECTURAL UNDERSTANDING TO RESOURCE AWARE SOFTWARE ARCHITECTURES REQUIRING DEVICE SYNCHRONIZATION

Chris Mattmann
University of Southern California
University Park Campus, Los Angeles, CA 90007
mattmann@usc.edu

Bilal Shaw
University of Southern California
University Park Campus, Los Angeles, CA 90007
bilalsha@usc.edu

ABSTRACT

We present a component-based software architecture that dynamically discovers and consumes remote services from distributed devices connected across a network. The architecture maintains its own local functionality, while also actively participating in its environment by discovering and responding to other devices as well. One novel capability of this software is its ability to synchronize its local and remote services with all other devices in its environment via its meta-architecture infrastructure. Furthermore, our architecture is fault tolerant and has the capability of re-synchronizing with lost connections and remembering old peers. The software architecture is built on top of the PRISM [1] middleware and inherits much of its design style from the C-2 [2] Architectural style.

KEYWORDS

Wireless Applications, Ubiquitous Computing, Interfaces.

1. PRISM MIDDLEWARE AND C-2 ARCHITECTURAL STYLE

The C-2 Architectural style specifies a topology of components, and connections between them [2]. C-2 applications are built from message passing components with no shared address space that share *connectors* between them. Connectors propagate 2 types of messages between their connected components:

1) A *Request* is a message that travels **up** the architecture that requests some type of functionality from the components that receive the message.
2) A *Notification* is a message that travels **down** the architecture that returns the results from the requested functionalities of the components.

The PRISM [1] framework extended the C-2 architectural style to support development of applications in distributed, mobile, and resource-constrained environments. PRISM adds peer to peer component messaging capability through *peer connectors*, and *peer messages.* PRISM also adds the ability to handle the **disconnected operation**, and a class of connectors called *Border Connectors* that communicate across networks to C-2 in order to facilitate its application to the aforementioned environments.

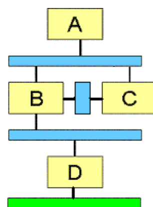

Figure 1. **A Sample Prism architecture with 4 components, labeled A,B,C,and D. The components are connected via two local connectors. Note that B and C are connected by a single *PeerConnector*. Also note that D is connected to a *BottomBorderConnector*, which spans device boundaries.**

Our work builds on the PRISM foundation and adds the meta-architecture through the implementation of the *Meta Component* which manages the *Border Connectors'* connection information, in addition to device services. All architectures that use our extension must have this *Meta Component*. The *Meta Component* manages local and remote

device information such as *Provided Services, Resource Locators*, and *Network Throughput*. We also introduce a *SynchDeviceThread*, which actively pursues lost connection information.

2. META-ARCHITECTURE AND DEVICE SYNCHRONIZATION

Meta-information, which is normally considered to be the burden of the implementing an application is a good candidate for being part of the architecture itself. This way when devices that are disconnected return to the environment, their functionality can be seamlessly re-integrated without any knowledge by the implementing application.

Table 1. Meta-architecture information stored in Device Table by each software application.

Device ID	IP Address	Service Port	CPU Speed	Network Throughput
D1	**127.99.92.225**	9009	85	35%
D2	**127.0.0.1**	9009	99	65%
D3	**23.23.33.222**	9009	91	75%
D4	**253.99.23.22**	9001	32	22%

We propose a ***Meta Component*** to store the meta-architecture information that all implementing PRISM software applications need to define. The Meta Component contains three tables of information:

1) A *Services Table*, which contains {deviceID,service$_1$, service$_2$, …service$_n$} tuples. The tuples map devices to provided services
2) A *Devices Table*, which contains {deviceID, Device Location. M_1,M_2 …M_n} tuples (where M_1..M_N is a vector of meta-architecture data that would be useful to store for each device. Examples of this are CPU Speed and Network Throughput). The tuples map devices to device meta-data.
3) A *Disconnected Devices Table*, which contains information about disconnected devices. The information stored in this table is the union of the *Services Table and the Devices Table*.

The addition of meta-architecture information helps us to seamlessly re-integrate disconnected devices into our system at the architecture level; however, it does not provide the low level functionality of actually marshalling the disconnected device metadata to the Meta Component for re-entry into the *Devices* table. We handle this problem by introducing another architectural level component, the *SynchDeviceThread*, which works in the following fashion. As each system is running, its own SynchDeviceThread will ping all devices in the *Device Table* every *n* seconds (where n is specified as a start-time parameter to the system). After the remote device is pinged, it is inserted into a **"soon to be disconnected"** queue in the *MetaComponent*. In essence, if a device comes into range and is pinged by the local calculator system, and it replies to the ping, then the local calculator system will check to see if the device that replied is in the current *Device Table* (multiple responses to pings are handled at the architectural level by PRISM using a message queue). If it is, then it must have been replying to an earlier ping, so it will be removed from the temporary disconnection queue if it is present there. If the device that replied is not in the *Device Table*, then it was possible that it was connected to the current device at a previous time, so we check the *Disconnected Device Table*. If it is in there, then its metadata is restored to the *Device Table* and the *Services Table.* The *SynchDeviceThread* also handles the case where a new device comes into the environment that has not yet been pinged—in this case it merely adds the device to the *Devices Table*.

REFERENCES

[1] Nenad Medvidovic and Marija Mikic-Rakic. "Programming-in-the-Many: A Software Engineering Paradigm for the 21st Century." *Workshop on New Visions for Software Design and Productivity*: *Research and Applications*, Nashville, Tennessee, December 2001.

[2] Nenad Medvidovic, 'Formal Definition of the Chrion-2 Software Architectural Style', Technical Report UCI-ICS-95-24, University of California, Irvine, Irvine, CA 92717-3425, (1995).