A Dynamic Platform for Runtime Adaptation

A Dynamic Platform for Runtime Adaptation

Hubert Pham,* Justin Mazzola Paluska,* Umar Saif,† Chris Stawarz,* Chris Terman* and Steve Ward*
*MIT CSAIL, Cambridge, MA, USA
{hubert,jmp,cstawarz,cjt,ward}@mit.edu
†Lahore University of Management Sciences, Lahore, Pakistan
umar@lums.edu.pk

***Abstract*—We present a middleware platform for assembling pervasive applications that demand fault-tolerance and adaptivity in distributed, dynamic environments. Unlike typical adaptive middleware approaches, in which sophisticated component model semantics are embedded into an existing, underlying platform (e.g., CORBA, COM, EJB), we propose a platform that imposes minimal constraints for greater flexibility. Such a tradeoff is advantageous when the platform is targeted by automatic code generators that inherently enforce correctness by construction.**

Applications are written as simple, single-threaded programs that assemble and monitor a set of distributed components. The approach decomposes applications into two distinct layers: (1) a distributed network of interconnected modules performing computations, and (2) constructor logic that assembles that network via a simple block-diagram construction API. The constructor logic subsequently monitors the configured system via a stream of high-level events, such as notifications of resource availability or failures, and consequently provides a convenient, centralized location for reconfiguration and debugging. The component network is optimized for performance, while the construction API is optimized for ease of assembly.

I. Introduction

Typical pervasive and mobile computing environments demand a degree of adaptivity beyond that supported by conventional application models. In order to maintain continuity of service, a mobile computing application must be able to discover and opportunistically exploit available resources, respond to component failures, and adapt its behavior according to the changes in its environment. While certain middleware platforms (e.g., CORBA, Microsoft .NET/DCOM, and Enterprise JavaBeans) provide a modular, component-based approach to application construction, the demand for adaptivity has motivated an additional layer of middleware, in the form of models that support adaptive applications (e.g., [1], [2]) as well as architectural description languages (ADLs) designed to explore, describe and verify run-time adaptive application architectures and styles ([3], [4], [5]).

Typical middleware models choose an underlying platform and extend it to a) support adaptivity and consequently b) impose a set of semantic constraints to ensure correctness ([6]). While this approach has value in the production and verification of reliable systems, the semantic framework and compile-time constraints imposed are designed to ensure the correctness of human-generated designs: an engineer is expected to grasp the model, devise a solution that fits the global semantic constraints, and interpret compile-time errors that guide his development of a certifiably correct implementation.

The value of a tightly-constrained semantic framework and compile-time verification becomes more questionable when the designs being processed are machine generated, e.g. by the goal-oriented planner of [7]. In the latter approach, source code is automatically generated at run time in response to run-time inputs, moving the generation of code from the human-centered development process to become an aspect of the run-time adaptation mechanism. Snippets of code are generated by scripted Techniques from an open-ended variety of sources, pieced together at run time to satisfy current goals, and executed on a middleware platform. There is no good mechanism at this point in the model to enforce global semantic constraints or to interpret compile-time error reports.

A more appropriate target middleware platform for automatically-generated designs is provided by run-time interpretive mechanisms (as opposed to compile-time, static mechanisms), substituting dynamic typing and minimalist semantics for strong models and verification tools. The designs produced by a planning process, and subsequently executed on the middleware platform, must be correct by construction, or those designs will fail. Because the planning process is entrusted with ensuring correctness in the code it generates, enforcing strong model semantics in the platform would be redundant at best, and at worst, potentially limit the adaptivity of the platform to that envisioned by the model. We propose that the middleware platform should be analogous to assembly language—unconstrained and flexible—leaving verification and correctness the concern of the compiler (i.e., runtime planning process) that generates the assembly code.

To address these needs, we have developed an adaptive middleware platform for application assembly that provides a target for applications whose structure is dynamically specified and reconfigured by run time processes. Rather than retrofit adaptivity into existing platforms, many of which inherently feature strong checking, we selected and implemented a set of lightweight mechanisms—network objects, liveness monitoring, composition, and hotswapping—to form a flexible, adaptive platform. Like many middleware models, our platform caters to fault-tolerance and adaptivity by promoting a strong separation between application-construction code and the underlying generic components that perform the computation. The constructor logic is responsible for assembling a system and monitoring its subsequent operation in a simple, single-threaded runtime environment. The system persists for as long as the application is active, reacting to runtime notifications of

978-1-4244-3304-9/09/$25.00 ©2009 IEEE

failures and newly-discovered resources, while the underlying modules may be opportunistically employed, replaced and adapted.

A. Design Goals

Our middleware platform targets applications that span heterogeneous devices, such as servers, desktops, handhelds, and mobile phones, and hence platform- and language-independence is a key requirement. In addition to supporting runtime adaptive behavior as described above, our work also strives to meet performance requirements: the runtime system must not impose any overhead on performance-critical computations and communication streams. Finally, our platform focuses on providing a dynamic runtime mechanism to *enable* late binding and reconfiguration, catering to rapid prototyping by either human developers or automatic planning logic. The platform, however, relies on the planning process to determine *when* to trigger adaptation.

B. Usage Scenarios

The main contribution of this work is a software platform for structuring adaptive, fault tolerant distributed applications designed to meet the above design requirements. There are several ways developers might use our platform:

- **Target for automatic code generation.** The motivating application for our framwork is its use as an "assembly language" by systems that generate code automatically or manage adaptation decisions based on context, such as our Goals Planner [7].
- **Library for prototyping adaptive applications.** Although motivated by the use of a goal-oriented planner as the constructor logic, our platform allows the rapid prototyping of adaptive applications via a simple top-level program that assembles components and monitors their subsequent operation. In such use, our work can be viewed as a scripting language alternative to environments with stronger semantics and more powerful verification tools: the platform provides enough constructs and mechanisms without imposing complex constraints or enforcing checks. With adequate testing, applications of reasonable size built using our platform can run on their own without additional middleware or verification.
- **Substrate for evaluating middleware models.** Our platform is simple enough to extend that it is potentially useful as a substrate for experimenting with sophisticated middleware component models. As many middleware systems traditionally bind their models to a specific language (e.g., Java) or a large component runtime (e.g., CORBA), our platform serves as a lightweight platform- and language-independent alternative.

We have used our platform both as a runtime substrate for the Goals Planner and to hand-build applications; Section IV describes our experience. Prior to that, Section II describes the platform in detail, and Section III overviews related work. Section V highlights implementation details, and Section VI presents micro-benchmarks. Finally, Section VII concludes.

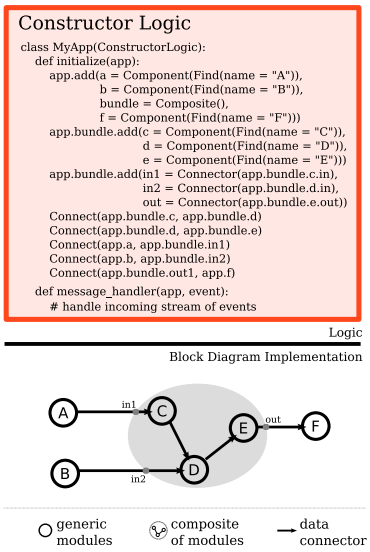

Fig. 1. Overview System Model. The constructor logic instantiates, configures, monitors, and potentially re-configures the underlying network of distributed components that collectively implement the desired application behavior.

II. The Platform

With our platform, applications divide into two layers: a *constructor logic* layer and a *component implementation* layer.

Synchronous constructor logic, produced either by automatic code generators or by hand during prototyping, instantiates, configures, and connects together a set of distributed, re-usable components, whose execution produces the desired application behavior. Once the component network is constructed and activated, the individual components run largely autonomously from the constructor logic. The communication paths between the distributed components are high-speed, asynchronous data links and do not incur the overhead associated with the synchronous communication mechanism used by the constructor logic to configure its components. The constructor logic monitors the operation of the resulting network via a serial stream of high-level events generated from and filtered within the component network, enabling it to reconfigure the running component network—or make other policy decisions—in response to environment changes, within a single-threaded environment. The relationship between the constructor logic and the components it monitors is depicted in Figure 1.

With our framework, it is natural to write applications that persist and adapt in the face of component failures because adaptivity policy is centralized. Additionally, the bi-level architecture separates various efficiency concerns and optimizes the form of efficiency that matters most to each layer. Operations in the component layer dictate run-time performance of the application; given our efficiency requirement, its exploitation of concurrency, multiple hosts, and arbitrary technologies cannot be restricted in ways that constrain performance. The constructor logic, on the other hand, is not directly

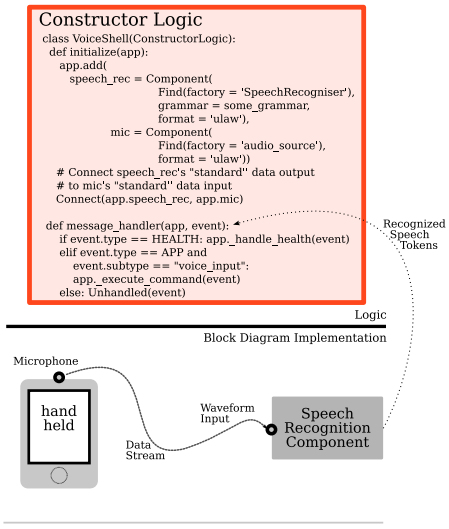

Fig. 2. Voice Shell distributed application.

performance-critical, and consequently may feature a simple single-threaded code environment for debugging, maintaining and reconfiguring component networks.

A. Construction API

This paper focuses on the construction logic and its interface to the component layer. Although there are many plausible semantic models for the constructor logic (e.g., generation of program text or use of language-specific reflection APIs), the constructor logic's block-diagram construction model is motivated by both a platform independence goal and the desire to provide a simple and versatile construction API. We strived to keep the library compact to minimize both the learning curve (for developers) and complexity (for code generators). Table I summarizes several library functions used by examples in this text. The choice of a block-diagram construction approach is inspired by ADLs and middleware models (e.g., [8], [9], [10], [4], [11], [3]) that express applications as component graphs, with adaptation as transformations on those graphs.

Code generators that target our platform first produce construction logic to build applications; they may subsequently adapt these applications by a) modifying the application structure via component hotswapping, or b) amending the constructor logic with additional event handling.

B. Example: Voice Shell

Figure 2 illustrates a simple yet real voice shell application, in which spoken user commands are translated to system behaviors. A user might say "play jazz" into a thin client; the utterance is sent to a speech recognition module on the local network, which in turn relays text representing the command.

To implement the voice shell, the construction logic 1) finds and instantiates both a Speech Recognizer component and an audio source (e.g., the microphone in a handheld), and 2) connects them together. Once constructed, the component graph runs autonomously and sends recognized tokens to the constructor logic for processing. The platform takes care of all the underlying details, providing essential mechanisms like data marshaling, health monitoring, and event handling, as will be described.

C. Component Modules

Our platform also provides developers with a rich API for developing component modules and managing their runtime life-cycle. We aimed to make it easy to create components that wrap existing off-the-shelf libraries and programs, such as a speech recognition engine (in the previous example), encouraging code re-use and rapid development.

Developers may also compose larger components from individual component modules. These composites of resources can then be monitored, replaced, and referenced just like other component resources. Composition reduces the tedium of managing and hotswapping instantiated resources, enables applications to exploit design hierarchy, and is central to scalability for supporting large, extensible application circuits.

Applying concepts from various ADLs ([12], [10], [13]) and hierarchical component models ([2], [3]), the basic building block in the platform is called a **Component**, an inspectable, distributed software module with communication ports. A **Pebble** is a primitive component and often manifests as a lightweight, policy-neutral module that implements a specific operation. A **Composite** is a collection of interconnected components (Pebbles or Composites) bundled together as a new component. Figure 1 depicts a Composite shaded in gray.

D. Essential Mechanisms

We identify and provide a set of practical mechanisms to construct a flexible and adaptive platform that handles the details of a distributed application runtime:

1) Network Objects: Network objects provide development flexibility (by allowing the location of each component to be specified independently of its function) as well as satisfy platform and language independence requirements (by translating language-specific APIs to language-independent network protocols). We wrap all components using lightweight objects called Network Portable Object Packaging (NPOP). NPOPs provide the veneer of a simple, sequential, and localized interface for controlling component networks, while reducing some development tedium with automatic stub generation. NPOPs are discussed in more detail in Section V-A.

2) Discovery & Monitoring: To find available components and adapt to changing resources, the platform runtime relies on two mechanisms for resource discovery and liveness monitoring. We employ `mDNS` [14] and `dns-sd` [15] for resource discovery and naming in a local peer-to-peer environment. However, since these protocols do not reliably detect device failures, the runtime automatically monitors liveness of all (distributed) components via a lightweight heart-beat mechanism, ensuring that applications will at least be aware of health changes in all dependency resources. The heartbeat service may be configured as a centralized resource to monitor components for all application instances (to avoid redundant connections).

TABLE I
SELECTED LIBRARY FUNCTIONS IN THE CONSTRUCTOR LOGIC API.

Function	Description
`Find(prop1=val1, prop2=val2, ...)`	Finds a set of matching factory components
`Subscribe(prop1=val1, prop2=val2, ...)`	Notify the application when components matching properties appear
`Component(factory, arg1, arg2, ...)`	Returns an instantiated component instance
`Connect(source, sink, ...)`	Connects two data-flow ports
`Composite(name = Connector(...), ...)`	Returns a new Composite with named ports
`Composite.add(name = Component, ...)`	Add a named component or connector to a Composite instance
`Composite.hotswap(old, new)`	Replaces an existing component with a new component instance

3) Event Queues: The platform provides a mechanism by which components, applications, and the liveness monitoring system can send and receive asynchronous notifications, or **Events**. In a large distributed application circuit, a large number of running components can give rise to a deluge of events from all levels of the component hierarchy. To effectively manage these messages, events are filtered, combined, and serialized before presentation to the constructor logic. Hence, planning processes typically need only monitor and reason about program behavior from a central location.

Because a Composite is typically aware of its constituent components' behaviors, the Composite collects and may filter events from its constituents before forwarding the message stream to its parent Composite. Developers define message filters, but a reasonable default (pass-through) is used if no filters are defined.

4) Hotswapping: The constructor logic reconfigures an activated graph by replacing existing or failed components. If a component is designed to support hotswapping, the platform runtime can orchestrate its replacement without disrupting the running component graph. Otherwise, the constructor logic is used to pause the component graph before manually conducting the replacement, which may temporarily disrupt service. To render a component hotswappable, developers must specify how to serialize (and de-serialize) a component's running state. The runtime performs minimal checking (e.g., it ensures that a replacement is pin-compatible with the original component) but purposely leaves the semantics to developers or code generators.

III. RELATED WORK

A. Remote Procedure Call, Remote Objects

Our model relies on our NPOPS package (Section V-A) for communication between distributed components. However, neither RPC nor remote object packages alone are sufficient to achieve highly reconfigurable and dynamic applications, as they typically assume relatively static hosts and connections. The BASE [16] architecture, on the other hand, explores micro-brokers for embedded devices in dynamic environments. BASE features a plug-in architecture to better shield the application developer from intricacies of the underlying network and closely resembles our network object mechanism.

B. Middleware Platforms

Our component model for application construction is reminiscent of the software component and the service oriented architecture paradigms. Related work in this family includes Sun Jini [17], CORBA CCM [18], [19], Enterprise Java Beans [20], IETF Service Location Protocol [21], Microsoft DCOM/.NET/WCF [22], and many others, especially those that involve web services. These packages are typically targeted at building highly-stable and relatively static enterprise-level applications from distributed components, whereas our work focuses on targeting adaptive applications within dynamic environments. Rather than retrofit adaptivity into existing frameworks, many of which inherently feature strong checking, our work explores a platform design that affords higher priority to runtime adaptivity and flexibility.

C. Reconfigurable Middleware and ADLs

While our middleware platform strives to be flexible by imposing few model-semantics constraints, we adopted several key ideas from numerous models.

Many projects in online application reconfiguration have inspired our block diagram construction approach, such as the Acme ADL [10], as well as C2 and Weaves [8]. C2 [23] composes applications in a hierarchy of independent components that communicate via asynchronous message passing. Weaves [24] is an object-flow architecture for building applications that process data flow. DoCoMo's DPRS [25] invokes a similar model for inspecting and upgrading mobile phone firmware.

Other middleware systems such as OpenORB [6], [26] focus on reflective component models as a prerequisite for adaptivity. While reflection (and introspection) are desirable features, our work notably differs from OpenORB's approach of embedding model constraints (e.g., predetermined meta-spaces) within the underlying middleware platform. OpenCOM [1] offers a flexible framework to assemble components at runtime but necessitates preexisting component binders and loaders to select and orchestrate the loading and unloading of modules. Our platform does not rely on binders or loaders (or their maintenance as the universe of components grows) to select or instantiate components, deferring that responsibility to the planning process.

In terms of adaptation approach, Plasma [3] and FASE [5] argue that adaptation policy decisions should be separate from generic components. Rainbow [13] provides formalism

to describe such adaptation policies. Fractal [2] articulates the need for component hierarchy to facilitate code re-use and sharing. Our platform's aspiration to separate application policy from mechanism is similar in spirit to aspect oriented programming ([27], [28]).

Several other ADLs explore formalism for ensuring adaptivity correctness. Taentzer [11] focuses on using a network of graph structures to manage and model dynamic changes in distributed applications. Additionally, Georgiadis et al. [29], like Cheng et al. [9] and the Aster project [4], explore and formalize automatic system reconfiguration from sets of formal constraints. Yau et al. [30] discuss and formalize a context-aware application model. Plastik [31] integrates ADL-based verification (using Acme) with the OpenCOM component model. We imagine that such systems could augment a code generator or planning process (that targets our platform) by providing the necessary algorithms for reasoning when to reconfigure applications.

Finally, other work in reconfigurable middleware include *dynamicTAO* [32], which focuses on fine-grain reconfiguration of the object broker and policies governing the runtime of the component (e.g., the broker's threading model), rather than on adaptation of the application as a whole.

D. Program Composition

Packages that specifically explore (block-diagram) program composition in a variety of domains also inspire our work, such as nesC [33] and Knit [34] for composing low-level operating systems components, as well as Gaia's LuaOrb [35] scripting language for controlling components within active spaces. Other projects, such as Ninja Paths [36], focus on automatically stringing together a linear set of wide-area, distributed software components to perform a necessary service. Our platform's main distinction from these projects is its support for both hierarchical component composition and runtime reconfiguration of component graphs.

E. Context-Awareness

Many contemporary component middleware packages, such as Aura [37], Gaia [38], Metaglue [39], Pico [40], PCOM [41] and Tuplespaces / Event Heap [42] focus on providing strong abstractions for building context-aware applications that capture and maintain user intent and/or data as users migrate between differing computing spaces or applications. We view our work as a potential underlying platform for such systems. Other packages, such as Speakeasy [43], optimize for peer-to-peer communications and require that all components export and ship adapter code to peer components to enable communication. Consequentially, Speakeasy components maintain control in order to arbitrate how they communicate with their peers. In contrast, our work trades component interface standardization for flexibility by maintaining all component control in the constructor logic rather than intermixed between the logic and component layers.

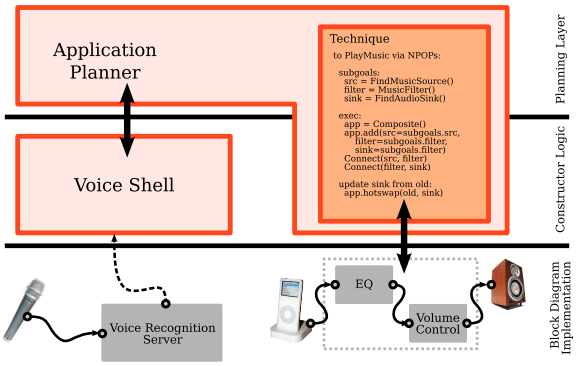

Fig. 3. The JustPlay [55] software environment builds a voice shell from components and connects the Voice Shell to a Planner [7]. If a user utters "Play Jazz", the Planner uses our runtime to construct a music-player application from several components and an audio filter Composite. It adapts the application by hotswapping components as the environment changes.

F. Data Stream Processing & Overlay Networks

Our platform's use of data streams between components is reminiscent of stream processing engines (e.g., Aurora [44], Medusa [45], Borealis [46]). These systems are typically geared towards supporting continuous real-time queries on large flows of data by applying data filters (chosen from a library of database-like primitives) at various points in the streams. They achieve fault tolerance through replication (potentially with configurable guarantees and trade-offs [47]). Unlike our platform, these systems typically target high-bandwidth, static environments with dedicated infrastructure. Like our work, Borealis supports dynamic modification of its data filters, but it does so by changing the filter's behaviors and parameters, rather than allowing graph reconfiguration.

While this paper is focused on our platform, our typical applications are also slightly reminiscent of stream based overlay networks (e.g., [48], [49], [50], along with many projects exploring stream processing engines), especially for media streams [51]. Our platform itself does not address the underlying routing path optimizations [52] for wide-scale data dissemination that these projects undertake, but applications using our platform can potentially apply many of these techniques.

Finally, while our system does not yet provide any mechanisms for measuring network bandwidth as a potential heuristic for adaptivity, several projects, such as Network-Sensitive Service Discovery [53] and Remos [54], focus on measurement and related techniques. Future work includes integrating these ideas with our platform.

IV. Applications

A. JustPlay

Our platform provides the foundation for JustPlay [55], a home automation application. JustPlay relieves the user of the task of configuring (and re-configuring) their home electronics. In a typical example, a user may say "Play jazz" to JustPlay,

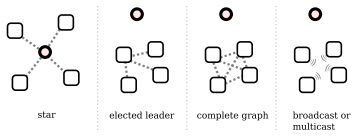

Fig. 4. Hub topologies chosen by the Hub component based on periodic network measurements.

and the system responds by searching for ways to play jazz given resources in the environment.

Architecturally, JustPlay is divided into two parts: a Goals Planner [7], which provides an extensible mechanism for managing system run-time decisions governed by user requests, and a set of components representing the devices and software available in the user's home. In response to user requests, the Planner asserts or revokes high-level Goals and searches the environment for Techniques that satisfy the user's Goals.

Each Technique provides a single part of a dynamically-generated constructor logic, typically a recipe for finding and connecting together resources found in the environment. Techniques may depend on other Techniques, so that one implementation of the `PlayMusic` Goal might use one component, while another might compose a large graph of components. For example, in Figure 3 the Technique finds and combines an audio source (e.g., an MP3 file server), an audio filter itself composed of several components, and speaker as one "PlayMusic" component. The Planner then passes the "PlayMusic" component to the voice shell for user interaction.

The Planner uses our platform's discovery, health monitoring, and object tracking mechanisms to keep track of the environment: when a component used by a Technique dies or when a better Technique becomes available, the Planner selects a replacement Technique and employs our runtime to hotswap in the necessary components for that new Technique.

Finally, JustPlay uses a variety of Pebble wrappers for off-the-shelf media streaming servers and players, such as Apple's Darwin Streaming Server (DSS) [56], the Apache Web Server, and VLC [57]. DSS streams data using RTP, while Apache streams over HTTP. Such wrappers enable the application Planner to discover, monitor, interconnect, and hotswap off-the-shelf software components with the constructor logic, hiding the complexities of the underlying transport protocols.

B. Hand-built Libraries

Overlay network data hubs that broadcast data streams are generally useful constructs within pervasive computing environments. For instance, Project Aura's IdeaLink [37], a collaborative blackboard application, and Gaia's ConChat [58], a contextual chat program, are examples of projects that rely on data hubs. As such, one useful building block in our applications is the Hub component.

Listing 1 illustrates constructor logic that first finds a data communication Hub module; as appropriate `ChatNode` Pebbles come online, the constructor logic adds them to the hub, enabling each node to broadcast messages to others.

```
from o2s.framework import *
class DataHubApp(ConstructorFramework):
    def initialize (app):
      # Subscribe to be notified when ChatNode Pebbles come online
      Subscribe(type = "ChatNode")
      # Find a hub component and instantiate it with the set of
      # participating nodes
      app.add(hub = Component(Find(factory = "Hub")))

    def message_handler(app, event):
      if event.type == NOTIFY:
            app.hub.add_nodes(event.components)
```

Listing 1. Constructor logic for instantiating a data-hub between several distributed nodes.

This application demonstrates the adaptivity possible with our platform: the Hub component itself can periodically perform a variety of network measurements among the participating nodes to determine the best communication topology (Figure 4). For instance, if the nodes are all relatively near the Hub, it may elect to implement a star topology in which all nodes communicate directly with the Hub. Alternatively, if the nodes are all relatively local to each other but far from the Hub component, the Hub may instead simply connect the nodes in a complete graph (or use network broadcast) to reduce communication overhead. The application designer need not worry about the particular (or best) topology, deferring the decision instead to the Hub component. The constructor logic remains relatively simple, while the Hub component itself encapsulates the implementation strategies.

We constructed a simple conferencing application using the adaptive Hub component, in which users engage in a conversation with other users. The layered approach of separating constructor logic from implementation enables the chat application to find and use the optimal set of devices available to each user in their respective environments during run-time. The application also maintains or upgrades active conversations when A/V devices fail or come into existence.

V. Implementation

Our platform is fully implemented in Python 2.5+,[1] along with prototype implementations in C and Java. It runs on nearly all major operating systems, as Python and Java are widely implemented; it has been tested on desktop machines running GNU/Linux, Windows, Mac OS X, as well as Familiar Linux [59] on iPAQ handheld computers and GNU/Linux on gumstix [60] embedded processors. Below we highlight certain implementation details:

A. NPOP: Network Portable Object Packaging

The NPOP framework provides a familiar RPC interface to object methods and state, and enables object references to be transparently passed among hosts. Each call on an NPOP takes arguments and returns values from a restricted set of types, including language-generic scalars, aggregates, and NPOP references.

[1] Source code and examples are available at http://pervasive.csail.mit.edu/.

In contrast to many traditional RPC packages such as Java RMI and CORBA where developers must generate client and server stubs for code modules—and potentially maintain a centralized registry containing such stubs—the NPOP abstraction uses a combination of dynamic stub generation and object reference tracking to hide both object locations and the distinction between local and remote objects. Additionally, the NPOP system ensures that the identity of a given NPOP instance is unique within an address space, allowing simple pointer comparisons to test for remote object identity. In our NPOP implementation, we employ XML-RPC [61] as the underlying RPC machinery, chosen for its lightweight implementation and transparency.

It is often necessary for NPOP objects to allocate resources on behalf of remote clients, as well as deallocate those resources when clients exit or become otherwise inoperable. To support garbage collection, the framework can optionally track the locations of an NPOP object's remote pointers and provide hooks for developers to handle resource (de-)allocation.

B. Pebbles and Composition

Pebbles are particularly well-suited for wrapping existing off-the-shelf libraries or programs. Developers create Pebbles by subclassing the `Pebble` class and overriding the relevant methods. Developers also specify the key-value pairs that a Pebble advertises on the network to indicate its presence and become discoverable. The runtime manages the life-cycle of Pebbles, providing hooks for developers to dictate how a given Pebble instantiates and initializes, runs (and optionally pauses), and shuts down.

Composites, hierarchical compositions of components, serve as an intermediary manager between their constituent components and the application specific logic (or a parent Composite). Composition and component hierarchy does not impose additional overhead in the runtime communication links between components. Once the constructor logic builds the component graph, the platform runtime computes and forms direct links between Pebbles, much like overlay networks.

C. Connectors

Typed data ports named **Connectors** provide components with standard input and output data streams: they serve as an interface to communicate with the outside world as well as an access point to the services offered by a component. Distributed components communicate by forming persistent **Connections** between pairs of connectors. Each connection is a typed, unidirectional communication path for typed data within an assembled application. A raw data byte stream that is emitted from an output connector is sent across the connection to the corresponding input connector, with support for side-band data. Data-flow on connections bypass all synchronous communication mechanisms used by the constructor logic to instantiate and maintain component networks; hence, connections do not incur the overhead of standard, synchronous RPC. Connectors are implemented as real sockets on Pebbles, but on Composites, connectors simply point to the next-hop socket endpoint.

Currently, socket-based connectors may use TCP, UDP, or RTP as the underlying transport protocol, but connectors can also wrap arbitrary off-the-shelf protocols. For instance, a popular way to watch remote media is to stream them over HTTP. With our platform, Pebbles can wrap the Apache web server and the VLC media player into discoverable components, and connectors can wrap Apache's and VLC's standard TCP sockets. One resulting benefit is that the construction logic can hotswap connections between Apache, VLC, or intermediate components.

D. Hotswapping Overview

Hotswapping provides a means to replace one running component with another, thereby facilitating service upgrade and fail-over. This section briefly discusses how the platform runtime hotswaps components without imposing any loss (or re-ordering) in the data streams between hotswapped components. The runtime can only seamlessly hotswap components if the component's designer anticipates and enables hotswappability by providing procedures to serialize component state. Otherwise, the constructor logic can hotswap via manual disconnection and reconnection of connectors, without data buffering.

As outlined in IBM's K42 project [62], the three main challenges to replacing a connected component during runtime are: 1) determining when to safely collect component state, 2) collecting and transferring that state, and 3) updating all external references to the old component. We summarize how we address these concerns:

- A Composite can hotswap any of its constituent parts with a pin-compatible component: the system enforces that the number and type of input/output connectors match between the hotswapped components. While in principle the system can perform static or runtime checking to also ensure a compatible API between the hotswapped components, our current implementation leaves this responsibility to the planning process.
- When a Composite hotswaps a constituent component, the Composite first instantiates the new (replacement) component. All new RPC calls to the old component are then blocked; in addition, all in-bound connections to the old component are atomically paused, so that the old component no longer receives any new data on its input connectors. Connector data now destined to the old component is automatically buffered on the sender side. The old component subsequently drains its in-flight data; once complete, the system assumes that the component is in a quiescent state, safe for hotswapping.
- The platform provides hooks in which developers implement state capture and restoration. For Composites, developers subclass the base `Composite` class and specify how to synthesize a state representation from the state of constituent components.

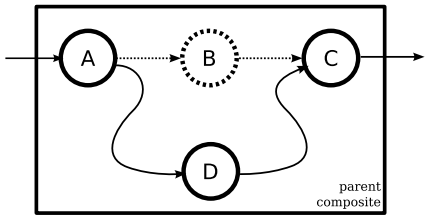

Fig. 5. Hotswapping Component B for Component D.

- We use the network object tracking mechanism to determine which components have references to the old hotswapped component and update those references.

E. Hotswapping Details

To facilitate hotswapping, every connector can internally sustain two concurrent connections—one *active* and one *passive*—but may only send data on the active connection. For input connectors, data received on the passive connection is buffered; only data received on the active connection is delivered to the component for processing. A connector can switch ("flip") the active and passive status of connections, in which the passive connection is promoted to active, and the (former) active connection disconnects and becomes passive.

Figure 5 illustrates hotswapping component B with D, without data loss or reordering. The platform performs the following procedures, orchestrated by the parent Composite:

1) Instantiate D.
2) Connect D's output connectors to C's (passive) input connectors.
3) Connect A's (passive) output connectors to D's input connectors. Note that data continues to flow through A's active connectors to B.
4) Atomically pause all of A's output connectors (and buffer any new data destined to B).
5) Inject a `DRAIN` token through B, which instructs B to drain its current inflight data. When the `DRAIN` token reaches C, C automatically flips its active and passive connectors.
6) Call `getstate()` on B, which blocks until a frozen state representation is available.
7) Call `start()` on D with B's frozen state as a parameter.
8) Flip A's active and passive connectors, and unpause, flushing all buffered data at A's output to D.
9) Reclaim B.

1) Discussion: The hotswapping procedure outlined above involves a mix of both RPC-based control layer mechanism, as well as connector level mechanism (e.g., `DRAIN` token). As such, special care is necessary because these two different mechanisms are not synchronized with each other. For example, because the `DRAIN` token is not synchronized with the activation of the new component (e.g., calling `start()` on D), as they respectively use a connector- and RPC-based mechanism, we buffer data from the new component if sent on passive connections. In other words, if C receives new data (from D) before B has fully drained (i.e., before C receives the relevant `DRAIN` token), C must buffer all such data until B has drained to ensure proper ordering.

We stamp `DRAIN` tokens with a nonce associated to a unique instance of a hotswapping procedure. When the orchestrating parent Composite forms a connection between D and C's passive connectors, the parent Composite also specifies the hotswapping instance nonce. This allows C to disambiguate between `DRAIN` tokens, in the case when there are multiple, concurrent hotswapping procedures.

There are several possible scenarios that govern the behavior of a Pebble (since only Pebbles have real connectors) when it receives a `DRAIN` token:

- If the Pebble has no passive connections, the Pebble is being hotswapped (e.g., B in Figure 5). It flushes and drains any inflight data, and forwards the `DRAIN` token onto *all* of its output connections. Doing so ensures that the `DRAIN` token will reach the output of the parent Composite, even if cycles exist within the network.
- Otherwise, if the Pebble's hotswapping instance nonce matches the token's stamp, the token signifies that the hotswapped component is now drained. The Pebble flips its passive and active connectors and discards the token. In practice, we set a timeout after which the Pebble automatically flips the passive and active connectors, even if no `DRAIN` token is received. This guards against the case in which a `DRAIN` token is lost due to component failure during the hotswapping operation.

2) Quiescence: In order to ensure and maintain quiescence during a hotswapping operation, we first ensure that there are no in-flight RPC calls to the old component and block new calls. After injecting the `DRAIN` token, we call `getstate()` on the old component, which blocks until it is safe to capture and return the component's state.

In the case of Pebbles, invoking `getstate()` will block until the Pebble receives a `DRAIN` token on one of its input connector. Once the Pebble has flushed its inflight data, it becomes safe to capture state. On Composites, calling `getstate()` will recursively call `getstate()` on all its constituent components. The Composite will wait until all `getstate()` calls on constituent components return, thereby collecting state for all of these components. The developer can specify how the Composite then synthesizes a new state encoding based on the collective constituent state.

3) Optimization: Many components often do not require state transfer; hence, components can elect to return immediately from `getstate()` if state transfer is unnecessary, thereby allowing instant activation of the replacement component to minimize switch-over delay. However, there is now a race between data flowing through the new component and the `DRAIN` token in the old component, so it is important to buffer data (e.g., at C) until the `DRAIN` token passes completely through the old component.

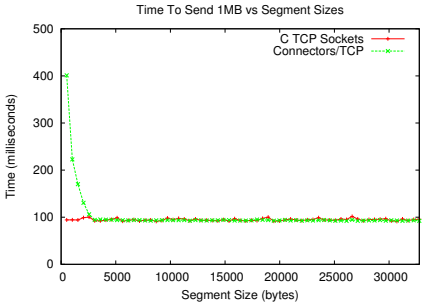

Fig. 6. Time to send a 1MB file between two hosts using TCP connectors.

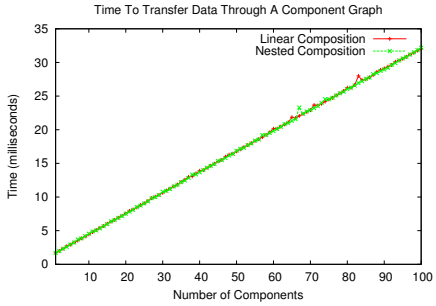

Fig. 7. Time to send a datum through a running graph; graph composition structure (linear vs recursive) does not impose runtime overhead.

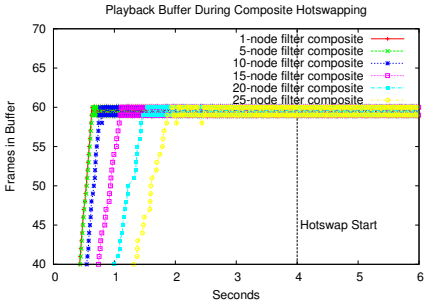

Fig. 8. Video frames (H.264, 10Mbps) in playback buffer as a 1–25-node Composite is hotswapped.

VI. Micro-benchmarks

Since the run-time performance of applications is dictated by the components and their connections, we focus the following micro-benchmarks on the components layer. The benchmarks below suggest that application networks suffer negligible additional overhead with our platform. The data streams test measures networking throughput between two hosts on the same 100 Mbit Ethernet subnet: a Pentium 4/512MB RAM and a Pentium 4HT/1GB RAM, both running Linux 2.6. For hotswapping measurements, we add: a PowerPC G4 Macintosh, 1GB RAM; an Intel Core Duo Macintosh, 1GB RAM; and an Intel Quad Core Macintosh, 3GB RAM.

a) Data Streams: Figure 6 plots the result of sending a 1MB file (filled with random bytes) between two hosts in various segment increments to simulate the stream-like process by which these modules would continually receive, process, and forward data to other modules. The objective of the benchmark is to compare the overhead introduced by connectors (using TCP as the underlying transport protocol). For each segment size, points are the average time of 50 runs.

As the data segment size increases, the amount of time attributed to platform overhead decreases. Our current implementation introduces some Python overhead compared to standard C sockets, but this cost is not architecture-imposed and reducible by implementing the connector infrastructure in a C module. However, because we typically send media segments in the range of 4-16K bytes, we find the Python implementation acceptable and on par with C sockets.

b) Composition: Figure 7 plots the time to send data through a Composite of varying structure and size. The first structure is a simple linear chain of N Pebbles. The other is one whose constituents include a Pebble connected to a Composite with $N-1$ components. Constituents are also structured similarly, achieving a deeply recursive hierarchy. All components run on the same host. In both structures, time grows linearly with the number of components. Performance is independent of structure, verifying that construction hierarchy incurs no additional performance penalty for data flow.

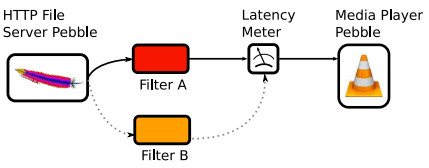

Fig. 9. Hotswapping performance measured as Filter A is hotswapped for Filter B.

c) Hotswapping: To measure the latency that hotswapping imposes on a typical application, we stream an MPEG-4 movie compressed with H.264 at 10 Mbps (a mid-range rate for compressed HD streams) from an HTTP server Pebble to a media player Pebble (Figure 9). Spliced within that path is a Composite implementing a stateless identity filter, which faithfully transmits data untouched, and a metering component which simulates a 2 second (60 frame) playback buffer while measuring its fluctuations. Each component runs on a unique host on the same subnet, and all underlying TCP sockets have `TCP_NODELAY` enabled to prevent packet batching.

In Figure 8, we measure the effects of hotswapping the filter Composite, as it comprises up to 25 linearly connected Pebbles. Prior to hotswapping, we allow the playback buffer to fill to capacity before commencing playback. The size of the Composite dictates the amount of time necessary to fill the buffer. Once filled, the playback buffer is depleted at the playback rate. At 4s after the media source begins streaming, we hotswap the filter Composite A for an identically structured Composite B. The hotswapping mechanism flushes the data within A (which will all enqueue in the playback buffer); the hotswapping latency of interest, measured at the metering component, is the time between when A is fully flushed and the arrival of the first video packet through B. As long as this time is shorter than the time needed to completely deplete the playback buffer, there are no playback artifacts. We observe that the playback buffer depletes by no more than 2 frames throughout the hotswapping process.

VII. Conclusion

Our platform provides a flexible and adaptive target for both automatic code generators and hand prototyping of pervasive applications. Its design is based on an ADL-inspired block-diagram abstraction used to separate a layer of simple constructor code from a network of interconnected components.

Our approach relies on the practical assumption of differing architectural priorities for these two layers: that the constructor code is not performance critical and should be easy to produce and modify, while the interconnected network of components cannot suffer significant performance overhead. The platform simplifies design of the constructor code by providing a simple and single-threaded context for its operation—but does not place restrictions on what kinds of architectures low-level components use to achieve their performance targets.

ACKNOWLEDGEMENTS

This work is sponsored by the T-Party Project, a joint research program between MIT and Quanta Computer Inc., Taiwan.

REFERENCES

[1] Clarke, M., Blair, G.S., Coulson, G., Parlavantzas, N.: An efficient component model for the construction of adaptive middleware. In: Middleware. (2001) 160–178
[2] Bruneton, E., Coupaye, T., Stefani, J.B.: Recursive and dynamic software composition with sharing. In: ECOOP. (2002)
[3] Serrano-Alvarado, P., Rouvoy, R., Merle, P.: Self-adaptive component-based transaction commit management. In: ARM, New York, NY, USA, ACM (2005)
[4] Bidan, C.: Aster: A framework for sound customization of distributed runtime systems. In: ICDCS. (1996) 586–593
[5] Allen, R., Douence, R., Garlan, D.: Specifying and Analyzing Dynamic Software Architectures. In: FASE. (1998)
[6] Blair, G.S., Coulson, G., Robin, P., Papathomas, M.: An architecture for next generation. In: Middleware, UK, Springer-Verlag (1998) 15–18
[7] Mazzola Paluska, J., Pham, H., Saif, U., Chau, G., Terman, C., Ward, S.: Structured decomposition of adaptive applications. In: PerCom. (2008)
[8] Oreizy, P., Gorlick, M., Taylor, R., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., Wolf, A.: An architecture-based approach to self-adaptive software. IEEE Intelligent Systems **14**(3) (May 1999) 54–62
[9] Cheng, S.W., Garlan, D., Schmerl, B.R., Sousa, J.P., Spitznagel, B., Steenkiste, P., Hu, N.: Software architecture-based adaptation for pervasive systems. In: ARCS. (2002) 67–82
[10] Garlan, D., Monroe, R.T., Wile, D.: Acme: architectural description of component-based systems. (2000) 47–67
[11] Taentzer, G., Goedicke, M., Meyer, T.: Dynamic accommodation of change: Automated architecture configuration of distributed systems. In: ASE. (1999) 287–290
[12] Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. **26**(1) (2000) 70–93
[13] Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: architecture-based self-adaptation with reusable infrastructure. Computer **37**(10) (Oct. 2004) 46–54
[14] IETF: Multicast DNS. Internet Draft, Cheshire & Krochmal (August 2006)
[15] IETF: DNS-based service discovery. Internet Draft, Cheshire & Krochmal (August 2006)
[16] Becker, C., Schiele, G., Gubbels, H., Rothermel, K.: BASE - A Micro-broker-based Middleware For Pervasive Computing. In: PerCom. (2003)
[17] Sun Microsystems: Jini component system. http://www.jini.org (2003)
[18] Object Management Group: The Common Object Request Broker: Architecture and Specification. 2.5 edn. (September 2001)
[19] OMG Corba CCM. http://www.omg.org/ technology/documents/formal/components.htm
[20] Sun Microsystems: Enterprise JavaBeans. http://java.sun.com/products/ejb/ (2003)
[21] IETF: Service location protocol, version 2. RFC 2608 (June 1999)
[22] Microsoft: .NET Framework. http://msdn.microsoft.com/netframework/
[23] Taylor, R.N., Medvidovic, N., Anderson, K.M., Jr., E.J.W., Robbins, J.E., Nies, K.A., Oreizy, P., Dubrow, D.L.: A component- and message-based architectural style for GUI software. Software Engineering **22**(6) (1996) 390–406
[24] Gorlick, M., Razouk, R.: Using weaves for software construction and analysis. In: International Conference on Software Engineering. (1991)
[25] Roman, M., Islam, N.: Dynamically programmable and reconfigurable middleware services. In: Middleware. (2004)
[26] Blair, G.S., Costa, F.M., Coulson, G., Duran, H.A., Parlavantzas, N., Delpiano, F., Dumant, B., Horn, F., Stefani, J.B.: The design of a resource-aware reflective middleware architecture. In: Reflection. (1999)
[27] David, P.C., Ledoux, T.: An aspect-oriented approach for developing self-adaptive fractal components. In: International Symposium on Software Composition. (2006)
[28] Pichler, R., Mezini, M.: On aspectualizing component models. Software Practice and Experience **33** (2003) 2003
[29] Georgiadis, I., Magee, J., Kramer, J.: Self-organising software architectures for distributed systems. In: WOSS. (2002)
[30] Yau, S.S., Karim, F.: An adaptive middleware for context-sensitive communications for real-time applications in ubiquitous computing environments. Real-Time Syst. **26**(1) (2004) 29–61
[31] Batista, T., Joolia, A., Coulson, G.: Managing dynamic reconfiguration in component-based systems. In: EWSA 2005. (2005) 1–17
[32] Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalhães, L.C., Campbell, R.H.: Monitoring, security, and dynamic configuration with the dynamictao reflective orb. In: Middleware. (April 2000)
[33] Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesc language: A holistic approach to networked embedded systems. In: PLDI. (2003) 1–11
[34] Reid, A., Flatt, M., Stoller, L., Lepreau, J., Eide, E.: Knit: component composition for systems software. In: OSDI. (2000)
[35] Cerqueira, R., Cassino, C., Ierusalimschy, R.: Dynamic component gluing across different componentware systems. In: DOA. (1999)
[36] Chandrasekaran, S., Madden, S., Ionescu, M.: Ninja paths: An architecture for composing services over wide area networks. Technical report, UC Berkeley (2000)
[37] Garlan, D., Siewiorek, D.P., Smailagic, A., Steenkiste, P.: Project Aura: Toward Distraction-Free Pervasive Computing. IEEE Pervasive Computing (April-June 2002) 22–31
[38] Roman, M., Hess, C., Cerqueria, R., Ranganathan, A., Campbell, R.H., Nahrstedt, K.: A middleware infrastructure for active spaces. IEEE Pervasive Computing (October-December 2002) 74–83
[39] Coen, M., Phillips, B., Warshawsky, N., Weisman, L., Peters, S., Finin, P.: Meeting the computational needs of intelligent environments: The metaglue system. In: Proceedings of MANSE. (1999)
[40] Kumar, M., Shirazi, B.A., Das, S.K., Sung, B.Y., Levine, D., Singhal, M.: Pico: A middleware framework for pervasive computing. IEEE Pervasive Computing **02**(3) (2003) 72–79
[41] Becker, C., Handte, M., Schiele, G., Rothermel, K.: PCOM - A Component System for Pervasive Computing. In: PerCom. (2004)
[42] Johanson, B., Fox, A.: The event heap: A coordination infrastructure for interactive workspaces. In: WMCSA. (2002)
[43] Edwards, W., Newman, M., Sedivy, J., Smith, T., Balfanz, D., Smetters, D., Wong, H., Izadi, S.: Using speakeasy for ad hoc peer-to-peer collaboration. In: CSCW. (2002)
[44] Abadi, D.J., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.B.: Aurora: A new model and architecture for data stream management. VLDB, 12(2) (2003)
[45] Zdonik, S.B., Stonebraker, M., Cherniack, M., Cetintemel, U., Balazinska, M., Balakrishnan, H.: The Aurora and Medusa Projects. IEEE Data Eng. Bull. **26** (2003) 3–10
[46] Abadi, D.J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A.S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.B.: The design of the borealis stream processing engine. In: CIDR. (2005)
[47] Balazinska, M., Balakrishnan, H., Madden, S., Stonebraker, M.: Fault-Tolerance in the Borealis Distributed Stream Processing System. In: SIGMOD. (June 2005)
[48] Andersen, D., Balakrishnan, H., Kaashoek, F., Morris, R.: Resilient overlay networks. In: SOSP. (2001)
[49] Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., etintemel, U., Xing, Y., Zdonik, S.: Scalable distributed stream processing. In: CIDR. (2003)
[50] Huebsch, R., Hellerstein, J.M., Boon, N.L., Loo, T., Shenker, S., Stoica, I.: Querying the internet with pier. In: VLDB. (September 2003)
[51] Zhang, X., Liu, J., Li, B., Yum, Y.S.P.: CoolStreaming/DONet: a data-driven overlay network for peer-to-peer live media streaming. In: INFOCOM. (2005)
[52] Pietzuch, P., Shneidman, J., Roussopoulos, M., Seltzer, M., Welsh, M.: Path optimization in stream-based overlay networks. Technical Report TR-26-04, Harvard (2004)
[53] Huang, A.C., Steenkiste, P.: Network-sensitive service discovery. In: USITS. (2003)
[54] Dinda, P.A.: Design, implementation, and performance of an extensible toolkit for resource prediction in distributed systems. IEEE Transactions on Parallel and Distributed Systems **17**(2) (2006) 160–173
[55] Mazzola Paluska, J., Pham, H., Saif, U., is Terman, C., Ward, S.: Reducing configuration overhead with goal-oriented programming. In: PerCom 2006: Works in Progress. (March 2006)
[56] Apple, Inc.: Darwin Streaming Server. http://developer.apple.com/opensource/server/streaming/
[57] VideoLan Player (VLC). http://www.videolan.org/
[58] Ranganathan, A., Campbell, R.H., Ravi, A., Mahajan, A.: Conchat: A context-aware chat program. IEEE Pervasive Computing **1**(3) (2002)
[59] The Familiar Project. http://familiar.handhelds.org/
[60] Gumstix Inc. http://www.gumstix.com/
[61] XML-RPC Specification. http://www.xmlrpc.com/spec
[62] Hui, K., Appavoo, J., Wisniewski, R., Auslander, M., Edelsohn, D., Gamsa, B., Krieger, O., Rosenburg, B., Stumm, M.: Supporting hot-swappable components for system software. In: HotOS. (2001)