An Ad Hoc Approach to Achieve Collaborative Computing with Pervasive Devices

Ren-Song Ko[1] and Matt W. Mutka[2]

[1] National Chung Cheng University, Department of Computer Science and Information Engineering,
Chia-Yi 621, Taiwan
korenson@cs.ccu.edu.tw
[2] Michigan State University, Department of Computer Science and Engineering,
East Lansing MI 48824-1226, USA
mutka@cse.msu.edu

Abstract. Limited computing resources may often cause poor performance and quality. To overcome these limitations, we introduce the idea of ad hoc systems, which may break the resource limitation and give mobile devices more potential usage. That is, several resource-limited devices may be combined as an ad hoc system to complete a complex computing task. We illustrate how the adaptive software framework, FRAME, may realize ad hoc systems by automatically distribute software to appropriate devices via the assembly process. We discuss the problem that ad hoc systems may be unstable under mobile computing environments since the participating devices may leave the ad hoc systems at their will. We also propose the reassembly process for this instability problem; i.e., assembly process will be re-invoked upon environmental changes. To further reduce the performance impact of reassembly, two approaches, partial reassembly and caching, are described. Our experimental results show that the caching improves performance by a factor of $7 \sim 40$.

1 Introduction

As technology improves, small devices and task-specific hardware begin to emerge. These devices usually have limited resources or specialized interfaces to address the desired goal of mobility and friendly usage. Thus, it will be a challenge to execute complex applications on these devices with reasonable performance and quality. However, the ubiquitous existence of computers may bring many possible solutions for this challenge. For instance, it is possible for computers to move and interact with their environment to seek the available resources to accomplish resource-intensive tasks more efficiently.

That is, instead of running software on a single device, one may look for available devices nearby and connect them together to form a temporarily organized system for short-term usage. Once the software is launched, the appropriate part of the code will be automatically distributed to each participating device. After that, these devices will execute the assigned code to accomplish the task collaboratively. Such a system without prior planning is called *ad hoc* [5].

C. Cérin and K.-C. Li (Eds.): GPC 2007, LNCS 4459, pp. 169–180, 2007.

Image the scenario that a person may watch a movie with his mobile phone. Because of limited computing capability, the video and audio quality may be unacceptable, and the viewing experience may not be pleasant. On the other hand, he may look for available intelligent devices nearby. For example, he may find an ATM machine for its larger screen and a MP3 player for its stereo sound quality. Thus, he may connect them together to form an ad hoc system as shown in Fig. 1. After the video playback software is launched, the appropriate part of the code will be distributed to each device, such as the code for audio processing to the MP3 player and the code for video processing to the ATM. As a consequence, instead of watching the movie on the mobile phone, he may enjoy the movie on the ad hoc system with larger image on the screen of the ATM and better sound on the MP3 player.

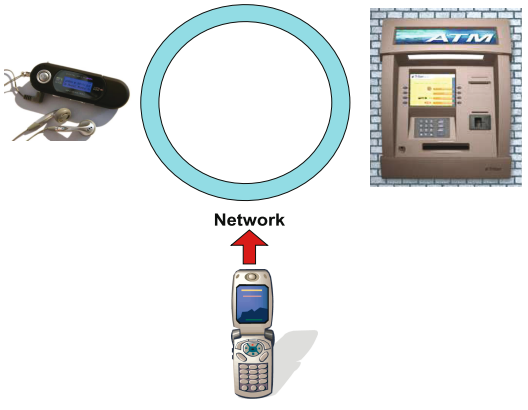

Fig. 1. A video playback application running on an ad hoc system

Such ad hoc systems may be realized by an adaptive Java software framework, `FRAME` [6, 7]. `FRAME` may automatically distribute software components to each participating devices and provide the functionalities of a middleware to allow these components to execute cooperatively. However, mobile computing environments are not likely static and, hence, ad hoc systems may be unstable. For example, some participating devices may leave the ad hoc system during the execution of the application. Therefore, the code on these leaving devices have to migrate to other devices in the system for proper execution of the application. In this paper, we shall illustrate the approach to improve `FRAME` for this challenge. We also discuss the issue of the performance impact on the application execution, and introduce two possible performance improvement.

We shall briefly describe the architecture of `FRAME` in the next section. Section 3 illustrates an approach for solving instability problem of ad hoc systems, discusses the performance issue, and describes how we improve it. We applied the improved `FRAME` to a robot application and measured the performance impact. The results

are illustrated in Sect. 4. Finally, the last two sections will give a summary, survey of related work, and then discuss potential future investigations.

2 Adaptive Software Framework: FRAME

The central themes of FRAME are component, constraint, and assembly. The architecture of FRAME [6, 7] may be summarized as follows.

Component: An application is composed of components. Each component provides services to cooperate with other components. The services define the dependency of the components and form a software hierarchy tree, i.e., a parent component requires services from its child components and vice versa.

Implementation: A component may have more than one implementation. Each implementation provides the same functionality of the component but with different performance, quality, and resource requirements. Only one implementation of each component is needed to execute a program. For example, the audio component of the video playback application may have two implementations. Each is able to process the audio of the movie but with different sound quality and computation resources. The implementation with better sound quality may require more computation resources than the mobile phone has. Of course, such an implementation should not be executed on the mobile phone. The question for which implementation is feasible on the given device will be answered with help from constraints. Finally, the software hierarchy information, such as what components the application has and what implementations of the component has, will be registered to a database server called the *component registry*.

Constraint: Each implementation may have a set of constraints embedded. A constraint is a predicate and used to specify whether the given computing environment has resources that the implementation requires. It may also specify the execution performance and quality of the implementation. The constraints of the implementation are used by the assembly process to determine whether the implementation is feasible on the given device.

Assembly: A process called *assembly* will resolve, on the fly by querying the component registry, what components and their implementations an application has. For each component, the assembly process will load each implementation and check its constraints on a given device. If all constraints are satisfied, the implementation is feasible on the device. Hence, the component with the feasible implementation will be distributed to the device. As shown in Fig. 2, there may be an implementation for audio component with better sound quality and all its constraints are satisfied on the MP3 player but not the mobile phone and the ATM. Thus the audio component will be distributed to the MP3 player.

Execution: After all the components are distributed, the application begin to execute.

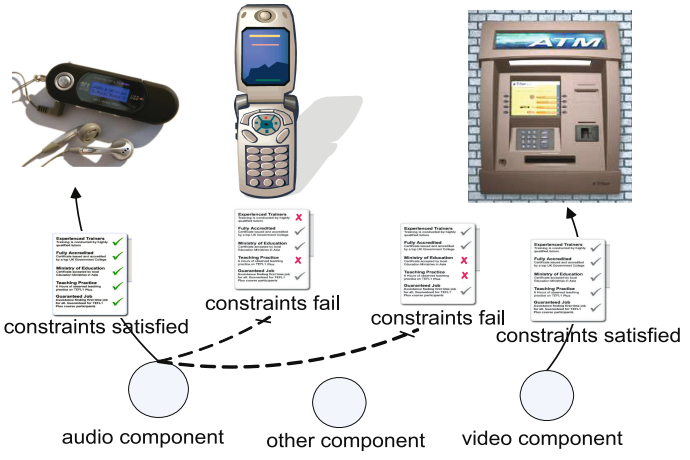

Fig. 2. Components will be distributed to appropriate devices based on their constraints

Table 1. `if-else` statement structure

```
if (constraints of component 1 with implementation 1)
{   // select component 1 with implementation 1

    if (constraints of component 2 with implementation 1)
    {   // select component 2 with implementation 1

        // check each implementation of component 3, 4,...
    }
    else if (constraints of component 2 with implementation 2)
    {   // select component 2 with implementation 2

        // check each implementation of component 3, 4,...
    }
    ... // more else if blocks for other implementations of component 2
}
else if (constraints of component 1 with implementation 2)
{   // select component 1 with implementation 2

    // similar as the code in the if block of
    // component 1 with implementation 1
}
... // more else if blocks for other implementations of component 1
```

The traditional approach to distribute components to appropriate devices based on constraints is to use condition statements such as if-else statements. For example, suppose there is an application that may have components $1, 2, \ldots, N$, where component i has M_i implementations. Thus, there may be nested if-else

statements similar to Table 1. First, it checks if the constraints of component 1 with implementation 1 are true. If yes, it will has code in its `if` block to check appropriate implementation of component 2, then 3, and so on. If not, it will jump to `else if` block to check the component 1 with implementation 2. The code in its `else if` block of implementation 2 are same as implementation 1. Thus, if constraints of implementation 2 are true, it will check appropriate implementation of component 2, then 3, and so on. The process will find an appropriate implementation for component 1 first, then 2, 3, and so on.

The condition statements approach is primitive from the software engineering perspective. As the number of components and their implementations increase, the code tends toward so called "spaghetti code" that has a complex and tangled control structure and the software will become more difficult to maintain or modify.

The most important limitation of the condition statements approach is that condition statements are hard-coded. Thus, the availability of all implementations need to be known during the development stage. It is not flexible enough to integrate newly developed implementations without rewriting and recompiling the code, and, of course, the down-time.

To avoid the above limitations, the assembly process uses the following two-step approach:

1. **Components distribution:** In this step, the assembly process will distribute components to participating devices. Note that there will be n^c different component distributions with n participating devices and c components. By using the information stored in the component registry, the assembly process may be able to identify all the component implementation of an application. Since the assembly process queries this information during runtime, the above limitations of the condition statements approach are avoided as long as newly developed implementations register their information to the component registry. When all components of a distribution are distributed, all the constraints will be collected and the assembly process will proceed to next step for solving these constraints.
2. **Constraints solving:** For each component, the assembly process will find out if all the constraints are satisfied. If all the constraints of the distribution are satisfied, the distribution is feasible and the application may execute on this distribution. `FRAME` uses a backtracking algorithm [8] for solving constraint satisfaction problems. If one of the constraints within this distribution is not satisfied, the assembly process will return to the first step for next distribution.

3 Reassembly

A straightforward idea for solving the instability problem of ad hoc systems is to monitor the computing environment changes. If some of constraints fail due to environmental changes, the application execution will be temporarily

suspended, the component assembly process will be re-invoked, and then the application execution will resume with appropriate implementations of the components. However, one challenge for this reassembly approach is performance, since the assembly process involves I/O activities, such as communication between devices, and intense computation, such as constraints solving to find the feasible distribution. In our experiments, the assembly process of the robot application is about 650 times slower than the similar application hard coded by if-else condition statements. It will be not feasible to simply re-invoke the assembly process for the reassembly, especially on a small temporal scale of environment change. Therefore, we propose two schemes, partial reassembly and caching, to improve the performance.

First, we observe that not all components need to be changed for the reassembly process and it is unnecessary to examine the constraints of these components. Thus, developers may only specify the subset of the components to be examined to reduce the run-time monitoring performance impact and the constraints solving time. For the video playback application example, the person may always carry the mobile phone and MP3 player, but not the ATM. As the person walks around, the connections between the ATM and other devices may drop, and then the ATM will leave the ad hoc system. Therefore, as shown in Fig. 3, it is only necessary to monitor the ATM and perform the video component migration when the ATM leaves.

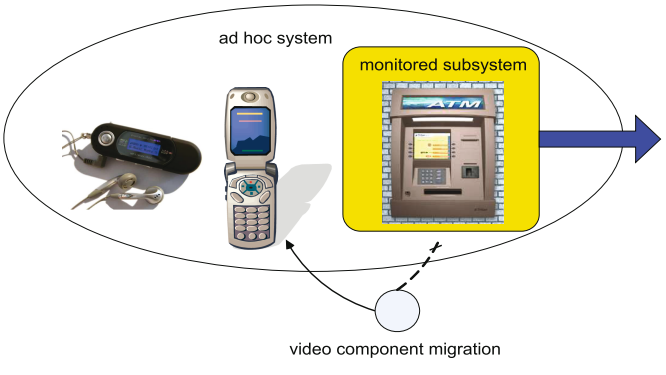

Fig. 3. Example of partial reassembly

The other performance improvement is to use cache, which may be done in two different levels. The first level is to cache the component distribution results, i.e., the first step of the assembly process. The purpose of the first step is to find possible distributions and collect all the constraints of each distribution for constraints solving. If no component implementation is added or removed, the constraints of each distribution will remain the same and the first step may be avoided.

The second level is to cache the computing environment, a more aggressive scheme based on the assumption that the computing environments will repeat.

A computing environment will be used as a key, and its assembly results are cached in a hash table with the key as shown in Fig. 4. That is, a computing environment may contain information that an application requires for execution, such as number of participating devices, network bandwidth, hardware, etc. Thus the information may be converted to a key for caching via a hash function. If the computing environment repeats, its assembly results may be obtained from the cache with the key.

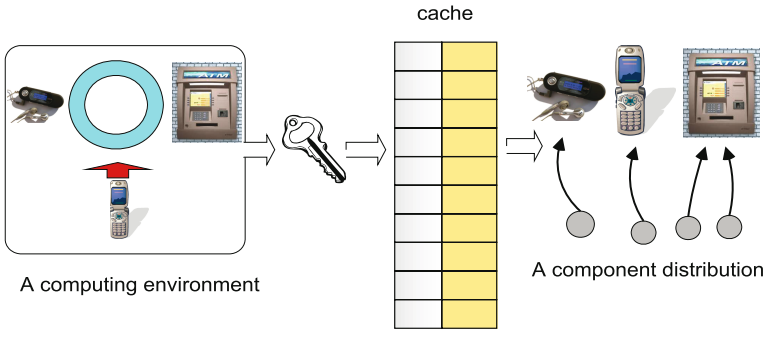

Fig. 4. Flow of reassembly cache

4 Performance Evaluation

We use a robot, XR4000 [12], to evaluate the performance of the component reassembly process. We compare the performance of different implementation selection schemes, including component reassembly with and without caching, and also evaluate the performance of the similar application using hard coded if-else condition statements. The performance is measured versus different number of the component implementations registered in the component registry.

To highlight the relationship between the performance and these different implementation selection schemes, we simplify the software hierarchy so the measured application has only one component with multiple implementations to be assembled. As a consequence, what the reassembly process actually does is to select an appropriate implementation of the component. Note that performance is application dependent, and, therefore, the performance comparison may not be same for different applications.

Figure 5 shows that the time required for the constraints solving step, which is about $50\% \sim 60\%$ of the total time for assembly or non-cached reassembly. If the application hierarchy does not change and no new implementation is added, the first level caching may be used. The non-cached reassembly may be approximately reduced to the constraint solving step, which is a $40\% \sim 50\%$ time saving.

Figure 6 compares the time required to search for the appropriate implementation of the component by the different schemes, i.e., non-cached reassembly, cached reassembly, and hard coded if-else statement. The if-else scheme requires about $0.003 \sim 0.018$ ms that depends on the number of implementations. The

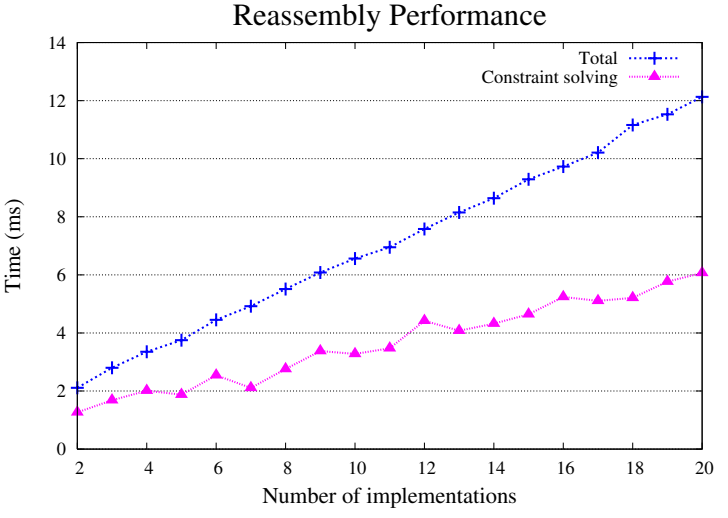

Fig. 5. Constraints solving performance of reassembly

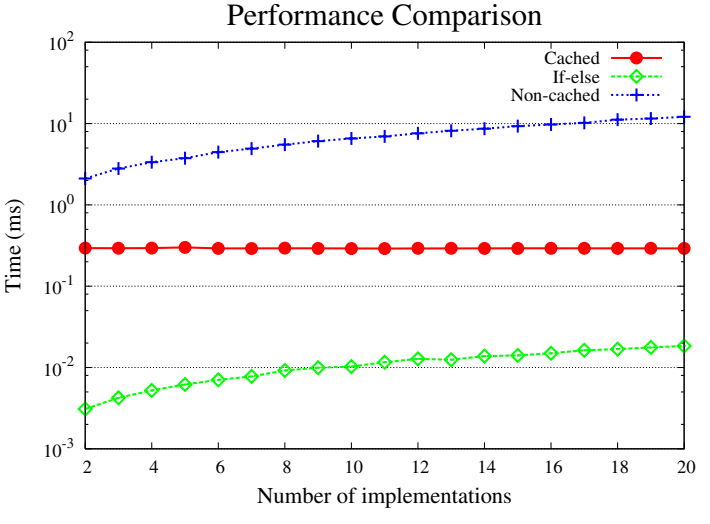

Fig. 6. Performance comparison for different component selection scheme

non-cached reassembly requires about 2.1 $\sim$ 12.1 ms that also depends on the number of implementations, and it is about 650 times slower than the if-else scheme.

The result also shows that the cached reassembly requires about 0.29 ms and improves the reassembly speed by a factor of 7 $\sim$ 40, and may be only about 15 times slower than if-else scheme. Unlike if-else and non-cached schemes, the cache

access time is constant and independent on the number of implementations. Thus, the performance improvement becomes more significant while the number of implementations increases. Also, the constant assembly time of cache makes the execution time of the application more predictable, which is an important issue for real-time applications.

Reassembly will load and unload the implementations of component whenever necessary, which will free some unnecessary memory, a scarce resource in embedded systems. Depending on how the application is developed, reassembly may save the memory usage. For example, the robot application using hard coded if-else statements has all implementations preloaded for better performance. However, this is a trade-off with memory usage. Fig. 7 shows that preloaded components require about 50% more memory than the reassembly scheme.

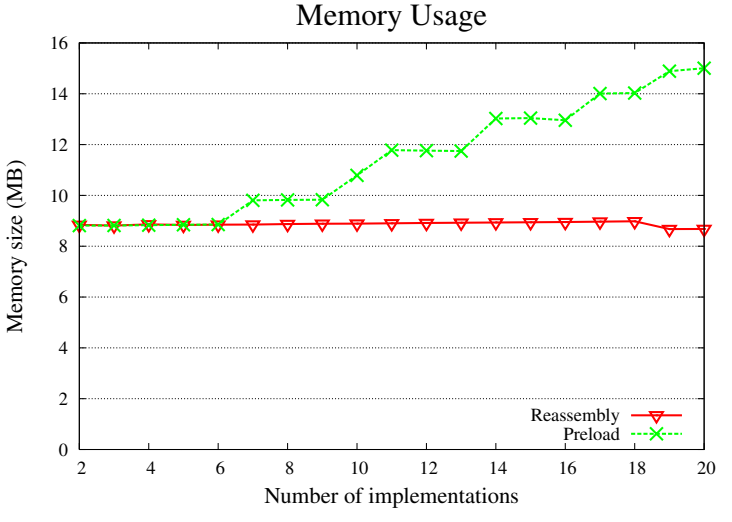

Fig. 7. Memory usage comparison for `ASAP` and component-preloaded

5 Related Work

The original idea of ad hoc systems is introduced in [5]. Lai, et al. [9] use infrared communication, which allows users to easily connect several devices as an ad hoc system via infrared communication. They also propose an approach to improve the performance of the assembly process by grouping the participating devices into "virtual subsystems" based on the hardware characteristics of the devices. With properly specifying the constraints, a component will only be distributed to the devices of the specified virtual subsystem and the time for the assembly process will be reduced.

There are several other related projects that may deliver applications on resource-limited devices and perform adaptation when necessary. The Spectra

project [2] monitors both application resource usage and the availability of resources in the environment, and dynamically determines how and where to execute application components. In making this determination, Spectra can generate a distributed execution plan to balance the competing goals of performance, energy conservation, and application quality.

Puppeteer [1] is a system for adapting component-based applications in mobile environments, which takes advantage of the exported interfaces of these applications and the structured nature of the documents they manipulate to perform adaptation without modifying the applications. The system is structured in a modular fashion, allowing easy addition of new applications and adaptation policies.

Gu, et al. [3] propose an adaptive offloading system that includes two key parts, a distributed offloading platform [11] and an offloading inference [4]. The system will dynamically partition the application and offload part of the application execution data to a powerful nearby surrogate. This allows delivery of the application in a pervasive computing environment without significant fidelity degradation.

Compositional adaptation exchanges algorithmic or structural system components with others that improve a program's fit to its current environment. With this approach, an application can add new behaviors after deployment. Compositional adaptation also enables dynamic recomposition of the software during execution. McKinley, et al. [10] gives a review of current technologies about compositional adaptation.

6 Conclusion and Future Work

Limited computing resources may often cause poor performance and quality. To overcome these limitations, we introduce the idea of ad hoc systems, which may break the resource limitation and give mobile devices more potential usage. That is, several resource-limited devices may be combined as an ad hoc system to complete a complex computing task. We also illustrate how the adaptive software framework, `FRAME`, may realize ad hoc systems. `FRAME` provides the functionalities of a middleware to allow software components to execute cooperatively. Most importantly, with constraints embedded in the component implementations, the assembly process of `FRAME` is able to automatically distribute these components to appropriate devices.

However, mobile computing environments are dynamic and ad hoc systems may be unstable since the participating devices may leave the ad hoc systems at their will. Thus, the code on some devices may need to migrate to another devices. We propose the reassembly process for this instability problem; i.e., if some constraints fail due to environmental changes, the application execution will be temporarily suspended, the component assembly process will be re-invoked, and then the application execution will resume with appropriate implementations of the components. Furthermore, the reassembly performance is an important issue for seemlessly execution of applications. To further reduce the performance

impact of the reassembly process, two approaches, partial reassembly and caching, are proposed. Our experimental results show that the caching improves the reassembly speed by a factor of $7 \sim 40$ and the time for reassembly is constant and hence predictable.

There is room for performance improvement. For instance, the constraints solving performance depends on the number of distributions and the number of constraints in each distribution. To improve the backtracking algorithm, if more information may be extracted from the relationship between constraints, some redundancy may be found between the constraints. Thus, truth checking for some constraints may be avoided. Moreover, more performance evaluation and measurement will be conducted in the future, including power consumption of large-scale ad hoc systems.

One important aspect of ubiquitous computing is the existence of disappearing hardware [13] that are mobile, have small form factor and usually limited computation resources. Since the constraints solving may require a lot of computation, these disappearing hardware may not have enough resources. One solution is to use a dedicated server for the off-site assembly process. Therefore, the participating devices may send the environment information to the server for assembly, and retrieve assembly results and the appropriate implementations of the components.

References

[1] E. de Lara, D. S. Wallach, and W. Zwaenepoel. Puppeteer: Component-based Adaptation for Mobile Computing. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, San Francisco, California, Mar. 2001.

[2] J. Flinn, S. Park, and M. Satyanarayanan. Balancing Performance, Energy, and Quality in Pervasive Computing. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, Vienna, Austria, July 2002.

[3] X. Gu, A. Messer, I. Greenberg, D. Milojicic, and K. Nahrstedt. Adaptive offloading for pervasive computing. *IEEE Pervasive Computing*, 3(3):66–73, July-September 2004.

[4] X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojicic. Adaptive Offloading Inference for Delivering Applications in Pervasive Computing Environments. In *Proceedings of IEEE International Conference on Pervasive Computing and Communications*, pages 107–114, 2003.

[5] R.-S. Ko. `ASAP` *for Developing Adaptive Software within Dynamic Heterogeneous Environments*. PhD thesis, Michigan State University, May 2003.

[6] R.-S. Ko and M. W. Mutka. Adaptive Soft Real-Time Java within Heterogeneous Environments. In *Proceedings of Tenth International Workshop on Parallel and Distributed Real-Time Systems*, Fort Lauderdale, Florida, Apr. 2002.

[7] R.-S. Ko and M. W. Mutka. `FRAME` for Achieving Performance Portability within Heterogeneous Environments. In *Proceedings of the 9th IEEE Conference on Engineering Computer Based Systems (ECBS)*, Lund University, Lund, SWEDEN, Apr. 2002.

[8] V. Kumar. Algorithms for Constraints Satisfaction problems: A Survey. *The AI Magazine, by the AAAI*, 13(1):32–44, 1992.

[9] C.-C. Lai, R.-S. Ko, and C.-K. Yen. Ad Hoc System : a Software Architecture for Ubiquitous Environment. In *Proceedings of the 12th ASIA-PACIFIC Software Engineering Conference*, Taipei, Taiwan, Dec. 2005.

[10] P. K. Mckinley, S. M. Sadjadi, E. P. Kasten, and B. H. Cheng. Composing Adaptive Software. *IEEE Computer*, 37(7), July 2004.

[11] A. Messer, I. Greenberg, P. Bernadat, D. S. Milojicic, D. Chen, T. J. Giuli, and X. Gu. Towards a Distributed Platform for Resource-Constrained Devices. In *Proceedings of the IEEE 22nd International Conference on Distributed Computing Systems*, pages 43–51, Vienna, Austria, 2002.

[12] Nomadic Technologies, Inc., Mountain View, CA. *Nomad XRDEV Software Manual*, Mar. 1999. Information available at *http://nomadic.sourceforge.net/production/manuals/xrdev-1.0.pdf.gz*.

[13] M. Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):66–75, Sept. 1991. Reprinted in IEEE Pervasive Computing, Jan-Mar 2002, pp. 19-25.