Spontaneous Container Services

A. Popovici, G. Alonso, T. Gross

Department of Computer Science
Swiss Federal Institute of Technology (ETHZ)
ETH Zentrum, CH-8092 Zürich, Switzerland
{popovici, alonso, trg}@inf.ethz.ch

Abstract. Container technology (e.g., Enterprise Java Beans) was designed for fixed network applications. This is unfortunate, because the ability of containers to adapt components transparently (e.g., with persistence and transactions) would be of great advantage in mobile computing. In this paper, we generalize the container model into a new software architecture, the *spontaneous container*. A spontaneous container allows to homogeneously extend all applications of a network, even if they spontaneously join or leave the network. In the paper we show how to build a spontaneous container by unifying different technologies into one coherent architecture: (i) dynamic aspect-oriented programming, (ii) containers, and (iii) infrastructures for mobile computing. Dynamic aspect-orientation makes a spontaneous container much more flexible than existing commercial containers. Inheriting the container programming model allows a single focus point for modifications for all applications in a network. Basing the overall architecture on dynamic service brokerage and discovery allows a seamless integration with existing infrastructures for both mobile and fixed computing. Following these ideas, we have built and evaluated a spontaneous container prototype that effectively and efficiently transforms applications within a network into a distributed system capable of transactional interaction, access control, and orthogonal persistence.

1 Introduction

Modern server architectures [14, 21] employ the *container model* to separate the business components from the system components. Through this separation, key functionality such as transactions, persistence, or security can be transparently added to the application at deployment time rather than having to implement it as part of the application.

Through separation of concerns [39], this model leads to increased re-usability and interoperability of business components. Although obviously very useful, this form of adaptation is not enough in software infrastructures for mobile computing where services appear and disappear arbitrarily and nodes cannot possibly know in advance with which other nodes they will interact. The frequent changes encountered in these new environments hamper the applicability of the container

model because deployment-time adaptation would require taking applications off-line before they can be adapted to a new mobile or ad-hoc network.

Instead of giving up the container model (and its benefits) for these dynamic environments, we advocate in this paper its generalization into a new software architecture for which we use the term *spontaneous container*. A spontaneous container provides a way to program and coordinate a number of entities in the same network – to have them working in a unified way in spite of their heterogeneity and their transient presence within the network boundaries.

From a software engineering point of view, the question is how to design and implement a spontaneous container. A promising way to achieve this objective is to provide explicit programming support for dynamic adaptation models [42, 20, 27] and aspect-orientation [9, 16]. By exposing the aspect-oriented run-time support in such a system, one gains the ability to unify at a small cost the programming paradigms encountered in (i) containers and in (ii) spontaneous networks for mobile computing. Thus, a spontaneous container inherits the properties of three technologies:

Spontaneous networks Spontaneous networks are characterized by dynamic service discovery [4, 19]. As a consequence, applications do not need static binding to external resources. Instead, these resources are located at run-time as the application moves from one wireless network to another. This way, arbitrary services can discover and use each other's resources while co-located in the same computing network. By computing environment, we mean any set of two or more applications that can interact with each other (e.g., using a wireless network).

Container programming The container model [22, 40] implies the ability to factor key functionality out of an application and make this functionality a dynamic property of the computing environment. I.e., rather than forcing the application to carry with it all the functionality necessary to interact with other applications or services, the computing environment of a spontaneous container should dynamically provide this functionality when needed.

Dynamic adaptation New programming support infrastructures [27, 34, 8] allow changing applications at run-time. This is the kind of property needed in a spontaneous container, where applications must be adapted on-the-fly, as they join or leave a given computing environment. The nature of these adaptations (transactions, orthogonal persistence, security, logging) suggests using the concepts of aspect-oriented programming. By *explicitly exposing* an adaptation interface for dynamic, aspect-oriented adaptations, one can achieve the level of flexibility needed by these highly dynamic environments.

In this paper we present a Java-based infrastructure capable of such run-time adaptation. Using this spontaneous container, nodes can dynamically acquire extensions that make their state persistent (the state may be stored at a base station or at another node), their interactions transactional (with arbitrary levels of nesting and interacting with either base stations or on a peer-to-peer basis), and subject them to an access control policy (when accessing information in other nodes or at base stations).

We first motivate the paper by providing two example scenarios that extend existing commercial systems and that we have used as our test-bed for the ideas presented (Section 2). In Section 3 we describe how to unify existing technologies into a coherent system architecture. We show how to implement this architecture in the context of applications running on a Java Virtual Machine (JVM) in Section 4. Based on this prototype, we have implemented network policies providing container managed persistence, access control, and transactional interaction. We explain how the container works by discussing every necessary extension step by step (Section 5). We also provide an extensive experimental study of the resulting system as a first step towards identifying the problems that need to be solved to make spontaneous information systems a reality. We discuss these results in Section 6 and conclude the paper in Section 7.

2 Motivation

2.1 Location services for mobile computing

Imagine a trade show or a large conference where participants are provided with computing devices such as desk-tops, lap-tops or PDAs (we will refer to these devices from now on as nodes) . Assume there is a spontaneous container that allows service publishing and discovery. Certain participants, like vendors and exhibitors use the fixed infrastructure. Other participants (e.g., visitors) use hand-held nodes. Nodes can communicate with either a base station or directly with each other. Using a spontaneous container, we want to provide the same services offered by a conventional middleware infrastructure. In particular, we would like to be able to provide the basic functionality found in a container (i.e., persistence, security and transactional interaction) but in a completely ad-hoc manner, that is, to all vendors registering at any time and to all visitors entering the fair grounds. Thus, nodes should be able to receive from the spontaneous container transparent functionality extensions and, after that, be capable of interacting transactionally among them, have their state persistently stored (in other nodes or a base station), and follow a simple access control policy. In addition to traditional service functionality (transactions, persistence, etc.), a spontaneous container may model location-specific features, such as service billing tailored to that particular fair ground.

In such a scenario, a spontaneous container provides the necessary flexibility. For example, if the access control policy of the organization evolves over time, it would be have new functionality extensions applied to all computing nodes within the organizational boundaries (in the example, a fair ground). On the other hand, if one computing node changes its location, all functionality extensions acquired within the boundaries of that location are discarded when the node leaves the area. Later on, when this node enters a new location provided with its own spontaneous container, it can be once again adapted at run-time with location-specific functionality.

As a refinement of this scenario, a spontaneous container could also be used to provide specialized policies for two distinct areas of the *same* fair ground. For

example, it can be used to increase the privacy of transactions in the sale areas of the fair ground by locally adapting nodes with the corresponding extensions, and to enforce a more permissive policy around meeting points.

2.2 Monitoring extensions for robotics

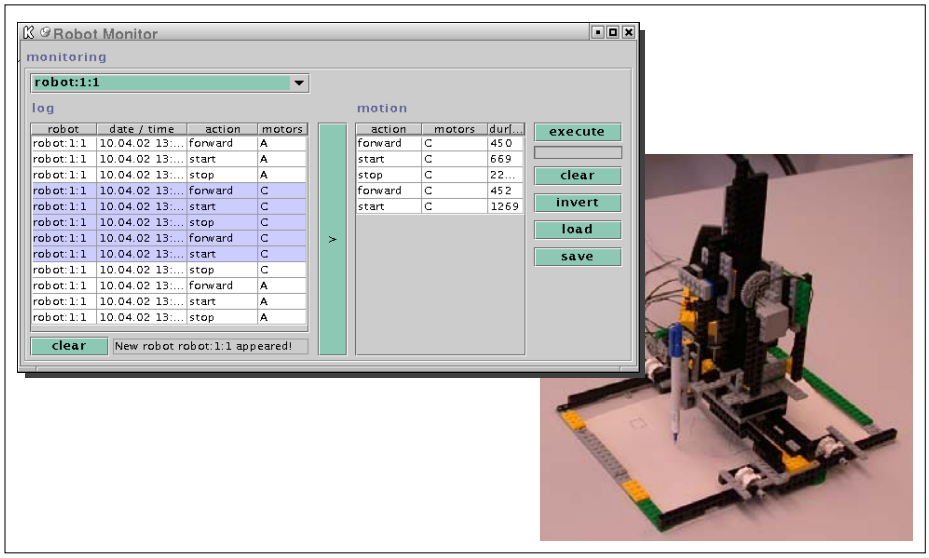


Fig. 1. A simple robot prototype (lower-right) transparently extended to show the motor actions in a graphical interface (upper-left).

Consider a manufacturing plant where a large number of mobile devices, robots and possibly "smart" artifacts collaborate to manufacture goods, control quality, receive production schedules and order sub-parts needed in the production process.

As an example, Figure 1 (lower-right) shows a simple prototype of a plotter that can be used in this scenario. When the plotter is moved into a given production hall, we want its behavior extended so that all relevant commands and service calls are logged in a database. The result is a comprehensive history of the relevant activities of the plotter (service calls, motor moves, sensor reads). All commands ever issued and executed within a specific context can be later analyzed, re-executed or undone. If a failure condition is detected, then a local base-station with a global view of all actions may trigger a compensation activity involving the coordinated movements of several devices. With the appropriate software support, the context could adapt the plotter so that the movements performed by the device are replicated and monitored on a nearby screen.

A variant is a robot (e.g., a transportation appliance) that enters a production hall. Assuming a well-defined geometrical model of the production hall, the robot does not need to learn by itself how to operate in this space. Instead, the robot's functionality can be adapted by the hall so its movements match the local working conditions.

A spontaneous container addresses these problems by treating all mobile devices that enter the production hall as components "deployed" within its scope. As such, they are extended on-the fly with the monitoring functionality.

It is important to notice that neither the robots nor the other applications are aware of the functionality extensions added by the spontaneous container. The extension can be added or removed as needed. If the robot is moved to a different location, that location's spontaneous container can add a new extension that indicates where the data must be sent for persistent storage. Or, within the same location, the extension can be exchanged for a new one that indicates that the data must be sent to a program that shows the movements in a graphic display, as shown in Figure 1 (upper-left).

3 Unifying technologies in the spontaneous container architecture

Spontaneous containers inherit the container programming model, dynamic resource usage and the ability to express cross-cutting run-time changes. This section shows how to unify these technologies into one coherent system architecture. In addition, we show how the resulting architecture differs from existing solutions in each of the originating areas.

3.1 Spontaneous container programming

A spontaneous container allows programmers to apply a homogeneous service policy to all entities of a networking area. We assume that every networking area is characterized by a spontaneous container policy P that describes the service model to be enforced on all entities within that network. An example of such a policy is "all remote invocations must be treated as transactions". This statement does not specify how to enforce, e.g., isolation, nor does it list the services and resources (database management systems, transaction monitors) accessible in the local network that will be used in the process of enforcing transactional behavior.

To enforce P, this declarative description may be translated into functionality extensions e_1 .. e_n. Each e_i contains concrete functionality to be added to all services within the boundaries of the considered spontaneous container. This functionality must describe what to change (e.g., what particular service invocations) and how to change it (e.g., how to bracket service invocations). Later on, if the policy is updated with a new version P^{new}, a new set of extensions $e_1^{new}...e_n^{new}$ may be generated and applied to the services within the network, while the obsolete extensions are revoked.

The spontaneous container programming model allows a single focus point for modification in both spatial and temporal dimensions. In the spatial dimension it addresses the increasing number of mobile computing devices that must be continuously adapted to the specific conditions of an enterprise, fair ground, production hall, etc. In the temporal dimension, the ability of a spontaneous container to deal with frequent policy changes in a uniform way addresses the continuous evolution of modern information systems. A clear advantage of this form of adaptability is that devices only need to carry their basic functionality. Any additional functionality is location specific and is inserted or extracted as need dictates. We create the spontaneous container in three steps.

3.2 Step 1: using dynamic AOP

In a spontaneous container one can create extensions that modify a running program. For this purpose it is first necessary to identify *where* to change the program. Potential points of interest may be, e.g., incoming and outgoing calls or variable access. Once a set of such points is identified, it is necessary to establish *what* additional actions are needed at those points. After this step, each time the execution reaches one of the points of interest, the execution is intercepted, and an additional piece of code is executed.

In a spontaneous container this information (where to change a program and what additional code to execute) is encapsulated into a single unit of software called *extension*. Typical examples of extensions are:

- Invoke `transactionBegin` before entering methods with names matching `"*transaction*"`.
- Invoke the additional code `updateDBTable` whenever a variable belonging to `"*EntityBean*"` classes is written.

Run-time adaptation is done by inserting extensions into applications. Extensions can also be withdrawn, leaving the program as if no insertion ever took place. Inserting and removing extension allows treating adaptations that affect a large number of points as a single modification operation.

The nature of extensions suggests employing aspect-orientation to program extensions. AOP allows factoring out of an application all orthogonal *concerns* (functionality) so that they can be treated separately [39]. Typical examples are distribution, security and logging when this functionality cuts across the system, i.e., it is not located in a single decomposition unit (e.g., class or package). Once this separation has been made, AOP techniques are used to combine the application with the orthogonal concerns when the application is compiled. This process is called *weaving* and is based on *crosscuts*, i.e., collections of points in the execution of a program where some additional functionality should be invoked. In AspectJ [43], for instance, these crosscuts could be the invocations of some method(s) of a set of classes.

Run-time changes to a program are usually performed in languages that explicitly support run-time adaptability such as composition filters [1] or reflection [15, 13, 24]. In the context of Java, reflective architectures such as MetaJava

[17], Guarana [25], or Iguana/J [36] can be used to support unanticipated software adaptation. Borrowing ideas from these techniques, frameworks for dynamic AOP have recently emerged [8, 26, 28].

Every node of the network carries with it the support for program modification (extension programming, extension insertion and extension withdrawal), which is explicitly available to all other nodes of a network. As we will show later, the support for adaptation in commercial platforms is either proprietary or not existent. By explicitly exposing this support one can express adaptations that could not have been foreseen at development or even at deployment time.

3.3 Step 2: extending container models with dynamic AOP

A good example of commercial container technology is the model included in the J2EE architectures [14]. This model, known as Enterprise Java Beans (EJB) [22] was the result of several years of evolution that lead to the separation of well understood middleware functionality from the business logic. The EJB specification describes *coding conventions* for creating business components. For example, all business logic methods must be previously defined in EJB-compliant interfaces; users must provide a number of administrative methods, prefixed by "`ejb`", as shown in Figure 2.a.

```
1.a class Account
2.a extends AccountEntityBean
3.a {
4.a     ejbActivate();
5.a     withdraw(float amount);
6.a }
```

(a)

```
1.b withdraw(float amount) {
2.b <identify remote caller>
3.b <authorize remote caller>
4.b <obtain a reference to a>
5.b a.ejbActivate();
6.b a.withdraw(amount);
7.b <end withdraw transaction>}
```

(b)

Fig. 2. (a) Business logic and (b) Wrapped business logic.

When developers follow this coding convention, the pure business logic can be *wrapped* in larger components that extend the business logic. The wrapping is done automatically at deployment time by the the server application called *EJB Container*. The EJB container typically generates code that deals with concerns that are orthogonal to the business logic. A simplified example of generated and deployed functionality is illustrated in Figure 2.b.

The container usage model implies that each site installs one or several containers on its site and configures this container to use the site-specific computing context (e.g., naming service, database servers). Then it acquires third-party

components and deploys these components in its own container. As explained, the container automatically adapts each component with additional functionality (lines 2.b-4.b,7.b), which indirectly reflects the site's computing environment.

The main *common* trait of both commercial containers and the spontaneous container architecture is the fact that a large number of points that often cut across the business logic components are automatically enhanced with orthogonal functionality. However, there are important *differences*.

The first one is the limited range of adaptations addressable by commercial containers. The kind of added functionality is "de facto" determined at development time: the coding conventions must be foreseen from the start in the business logic components. In addition, commercial containers typically use a proprietary technology for adding functionality, tailored for usage with the considered coding conventions. The added functionality corresponds exactly to the middleware features foreseen in the container specification. This limitation becomes quickly unacceptable when trying to add site-specific features. In practice, this would require the ability to change the often proprietary container implementation (resulting in a lack of tailorability of container models [29]). By contrast, a spontaneous container provides explicit support for adaptations located at the run-time environment level. This more general approach allows for adaptations that could not have been foreseen at development time. Through this feature, a spontaneous container does not require applications to be developed using a specific coding standard.

The second important difference is the limited flexibility of commercial containers – they add functionality at deployment time. The reason is that containers have been built for deploying long-running server-side business components. Once deployed in a commercial container, a business application will use the *same* computing context (nearby database management servers, naming services, etc.) throughout its lifetime (or until these resources are updated). The frequent changes encountered in mobile and embedded computing do not fit in this model and are difficult to address at deployment time. The explicit support for run-time adaptation makes a spontaneous container more appropriate for these environments.

3.4 Step 3: extending spontaneous networks with dynamic AOP

Spontaneous networks [19, 4] are systems for service discovery and brokerage. They provide the flexibility needed in mobile computing: applications do not know from the start which other services they are going to use throughout their lifetime. When a computing node enters a community of services (e.g., by joining an ad-hoc network) it *discovers* the available resources (nearby services for fair exchange, stable storage, light switches, printers, message boards, etc.). It uses these services through their published interfaces until it leaves the considered network. Later on, it may join a new community of services where it discovers and uses a new set of services.

Figure 3.a illustrates the basic mechanism used to discover and use new services in Jini [4], a platform for spontaneous networking. A service (B) joins

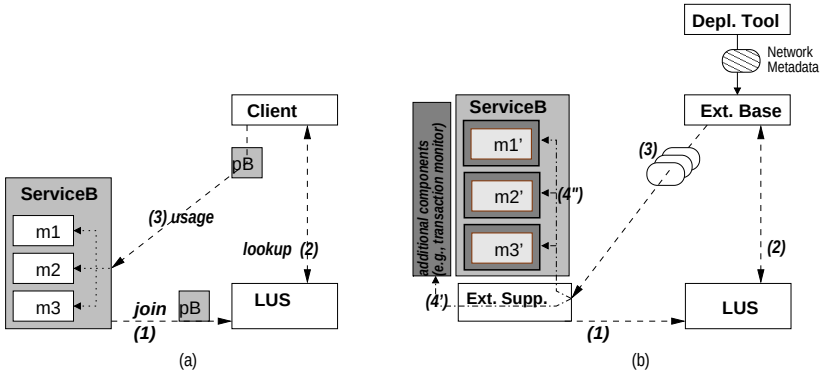


Fig. 3. (a) Typical interactions in Jini and (b) Run-time adaptation of application through a Jini interface.

the computing environment by registering a proxy pB at a nearby lookup service (step 1). The proxy contains code and data that can be moved freely between nodes. Other nodes can query the Lookup Service (LUS) (step 2), obtain pB and use the service B through its proxy pB (step 3). For instance, the call $pB.m1()$ will result in the execution of $B.m1()$. Alternatively, nodes can ask to be notified when a service matching a certain template joins or leaves the Jini community.

The main advantage of this type of spontaneous networks over container platforms is that resources are discovered and used at run-time. Because of the mobility constraint, the functionality needed for transactions, security, authentication, orthogonal data persistence is hard-coded during the development of mobile applications. This functionality must be available beforehand, even if internally it may use resources discovered at run-time. Thus, the main drawback of spontaneous networks is the lack of support for separation of concerns: they miss the properties of the container model (where business logic is reusable, while service functionality can be separately added at deployment time). This fact can cause several problems. For instance, two mobile services developed with different transaction models in mind are not inter-operable. This could not happen in a container model, where the transaction logic is uniformly added to all components by the same container. In addition, the enforcement of local policies (e.g., usage of a certain degree of encryption for all communications) is also difficult to achieve.

A spontaneous container inherits from spontaneous networks the ability to dynamically discover and use other services, but overcomes the shortcoming of hard-coded service functionality by exposing the support for dynamic aspect-oriented programming.

To achieve this goal, the spontaneous container architecture extends spontaneous networks as follows. It defines an adaptation service that allows the uploading of extension objects into a node. Each node must carry an adaptation

service, which is accessible as a regular service. Figure 3.b illustrates service B running together with an adaptation service (marked in the figure with "Ext. Supp.") on the same JVM. Like any other service, the adaptation service joins the spontaneous network community (step 1) and allows other nodes to use its interface. The extension base is continuously scanning the network for new adaptation services (step 2). Once a new adaptation service is discovered, a customized set of extensions objects is sent to it (step 3). When receiving the extensions, the adaptation service instantiates in step (4') the additional sub-components received through extensions (e.g., transaction monitors, access control modules, encryption components) and then it adapts in step (4") the existing methods $m1$, $m2$, and $m3$, such that they execute additional functionality at specific joinpoints. Step (4") is performed through the explicit aspect-oriented support of the adaptation service.

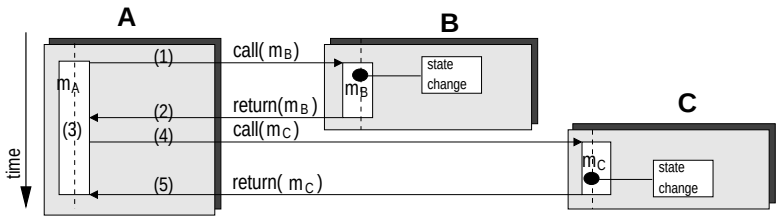
4 Building spontaneous containers for Java environments

To test these ideas, we have created a spontaneous container prototype for Java services. We created the prototype by filling in the architectural framework with concrete components for run-time adaptation, container programming and dynamic service discovery. For run-time changes we use PROSE [34], an aspect-oriented run-time system developed for this purpose. We add container-specific features by extending PROSE to perform container-like adaptations. Finally, we use Jini for dynamic service management and discovery so that every node can exchange, provide, and receive extensions at run-time, thereby providing the basic mechanism for spontaneous interaction.

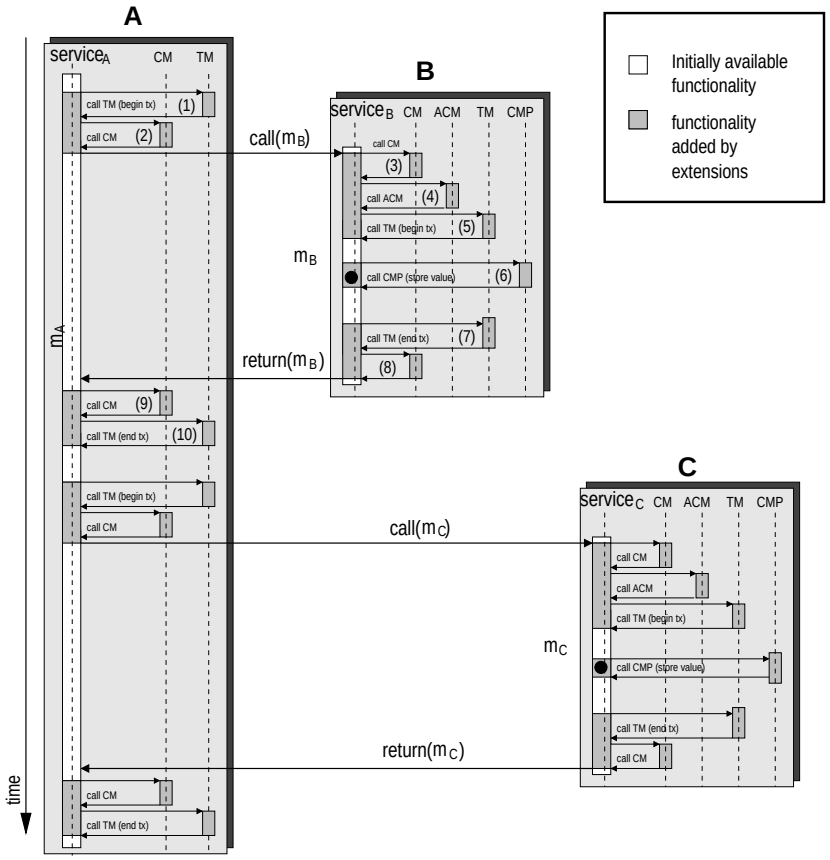
4.1 How extensions work

Consider three nodes, A, B, and C that want to interact as follows. Node A initiates a distributed computation within method m_A (Figure 4.a). It first performs some local operations and then invokes method m_B on remote service B (step 1). The computation in m_B changes the state of the service B at a place denoted in the figure by ($\bullet$). m_B completes, and the results are transferred back to A (step 2) where additional operations are performed locally by A (step 3). The remaining code of m_A involves a second remote call to m_C, carried out in a similar manner (steps 4 and 5). Assume now that these nodes have acquired extensions that will enhance the computation just described with context management (CM), transaction management (TM), container managed persistence (CMP) and access control (ACM).

Figure 4.b shows the control flow once the extensions are in place. We indicate with gray shading that the control flow passes through functionality added by extensions, while the white parts correspond to functionality initially available in the methods m_A, m_B, and m_C. The time axes are not drawn to scale. As before, the computation starts in m_A. When the remote invocation to m_B is



(a)



(b)

Fig. 4. (a) Initial control flow and (b) Control flow after extensions have been added.

initiated, the TM extension glue traps the call (step 1). The extension glue invokes the TM functionality to create the necessary transactional context (e.g., a transaction identifier). The transactional context and A's identity are transmitted to B as implicit context (since they do not exist in the signature of m_B). The CM extension located in the communication layer, marshals the implicit context data together with the parameters of the call (step 2), before the invocation takes place.

At node B, the local CM extension detaches the context data (3) and associates to it the current thread of execution. Before the application logic in m_B starts executing, a number of things happen. First, the ACM extension is invoked to check whether A has the right to access m_B (step 4). If access is granted, the TM extension of B is invoked to keep track of the transactional context and to start m_B as a local transaction (step 5). During the execution of m_B, a state change occurs ($\bullet$). The CMP extension glue intercepts the state change and notifies the object-relation mapper (step 6). The mapper will forward the update to the database. When the execution of m_B is completed, the TM extension is invoked once again (step 7) to pre-commit the changes carried out in step 6. If TM at B produces any context information that must be shipped back to A, the CM extension marshals this data together with the return of the call (step 8). At A, any context information is extracted from the return values (step 9), and passed to the TM extension for processing (step 10). After that, the execution of m_A resumes with analogous steps for the call to m_C.

4.2 How to program extensions

We are interested in inserting extensions in *running* Java applications. Thus, we use PROSE [34], a JVM extension that can perform interceptions at run-time. However, an application will not see any difference with a standard JVM. The PROSE JVM also provides an API for inserting and removing extensions.

PROSE extensions are regular Java objects that can be sent to and be received from remote nodes. Signatures are used to guarantee their integrity. Once an extension has been inserted in the JVM, any occurrence of the events of interest results in the execution of the corresponding extension. If an extension is withdrawn from the JVM, the extension code is discarded and the corresponding interception(s) will no longer take place. We do not discuss the PROSE architecture [34, 33] in detail, but we explain how such extensions can be created in the context of a spontaneous container. There are two basic mechanisms for expressing run-time changes: (1) by directly programming PROSE extensions in Java or (2) by generating PROSE extensions from declarative deployment descriptors.

(1) Programming PROSE extensions Figure 5 contains an example of a PROSE extension. We do not get into the details of extension coding, but this example should give a first impression of the mechanisms used. A more detailed description of PROSE extensions can be found in [34].

```
1 class BeforeRemoteCall extends MethodCut() {
2      // if we are the root, create a transactional context
3      public void ANYMETHOD(ServiceB thisObj, REST params)  {
4              if ( lookup(TX_CONTEXT) == null) {
5                      txCtx = ROOT_CTX;
6                      Transaction.bind(currentThread(),txCtx);
7                  }
8          }
9       { setSpecializer(ClasseS.extending(Remote.class)). AND
10                      (MethodS.BEFORE) ); }
11   }
12 }
```

Fig. 5. A Java class containing the code for a run-time extension.

The extension is called `BeforeRemoteCall`. It defines what to do (transform an invocation into a transaction) and where to apply this action (before incoming remote calls of services of type *ServiceB*).

The method on line 3 instructs PROSE to execute its body for every method invocation (wildcard `ANYMETHOD`) of services of type `ServiceB`. The predefined class `REST` is a wildcard that denotes arbitrary parameters list. Thus, the special signature matches invocations of the form `ServiceB.*(*)`. This is however not enough, as we want transaction to be activated just before remote calls. This specialization is achieved on line 10 where a combination of building blocks specifies that services should extend the `Remote` class (thus, only methods of remote services will be intercepted by PROSE) and that the extension should be executed before the actual business logic in that method (through `MethodS.BEFORE`).

The body of this method represents the extension action to be executed at all code locations that match the conditions described above, that is, before incoming remote calls from other services. The extension instructs the transaction management component to create a transaction context if the current invocation is not within the scope of a transaction already (line 4). It then associates the newly created transaction with the current thread of execution. This way, all subsequent invocation to other services will be part of the newly created transaction. To activate this extension, an object of type `BeforeRemoteCall` must be passed to PROSE. This is done using the `PROSE.addExtension` method.

In PROSE it is possible to perform more accurate and complex interceptions than those used in this example. For instance, it is also possible to intercept method calls based on the type of parameters passed. The range of the interception can be widened by using wildcards, since in an ad-hoc environment the methods are not necessarily known in advance. For scenarios where more knowledge about the application is available, the interceptions can be made much more precise, by including method signatures, parameter types, etc.

An example of a very useful extension that can be written without knowing the source code is an encryption extension. Such an extension would intercept any incoming or outgoing RMI call to an application and perform the necessary

encryption or decryption. Extensions can also be written knowing only the published interface of an application by using the method name, the class name, the signature, or even the parameters to specify where to intercept the execution. An example of what can be done using this information would be a logging extension that creates a log record in some remote database every time a given service is called. Finally, extensions can be written with full knowledge of the source code.

(2) Describing policies declaratively To fully support the container programming model, policies applied to all entities of a network are described declaratively, usually as XML data. This has been done by extending PROSE with support for the equivalent of the *deployment descriptors* used in EJB architectures [22]. This support comes in the form of a meta-data repository (network meta-data in Figure 3.b) that specifies, in a declarative fashion, what types of services are expected to be adapted and in which way. For instance, it might say that the encryption extension is to trap all RMI calls and encrypt them; a QoS extension is to trap outgoing RMI calls and cancel them if there is not enough bandwidth available; a load balancing extension is to send requests to different servers as dictated by the current load; a billing extension is to trap calls to a specific service and generate a charge for its use.

This information, which needs to be provided by the system's programmer, is used by the *extension base* to generate extensions from these specifications. The resulting PROSE *extension object* [34] can be sent through the network to all nodes that need to be adapted.

4.3 Acquiring and discarding extensions

In our architecture, all nodes fall into two categories. *Extension base* nodes contain a database of extensions. They discover new nodes joining the network and send extensions to the newcomers. *Extension receivers* carry the component that receives extensions from extension bases. We assume that each extension receiver has PROSE activated on its JVM. When it obtains an extension from an extension base, it immediately inserts the extension using the PROSE API. Extension receivers also discard extensions when they leave a network or lose contact with the extension base.

By appropriately assigning extension base and extension receiver roles, one can achieve adaptations of various service communities. At one extreme, each node can contain an extension base. When it joins a new community, it distributes its extensions and receives others from the existing nodes. This peer-to-peer organization is appropriate for creating an information system infrastructure in entirely ad-hoc communities. At the other extreme, each physical location (e.g., a fair ground) may have a base station as extension base. All other nodes (e.g., the mobile nodes) are extension receivers. This organization is appropriate for adaptations that correspond to infrastructure and organizational requirements. Between the two extremes, many other configurations are possible.

When considering extension objects, we distinguish between two kinds of adaptations. These adaptations use the same PROSE mechanisms but they play different roles. The first kind is the extension *functionality*. The second kind is needed to intercept events of the application and connect the application logic to the extension functionality. We denote this second type of adaptation extension *glue* and the points where it must be added *join-points*. Every extension we employ contains a functionality and a glue part.

We used the leasing mechanism provided by Jini [4]: the extension objects sent to each node are actually *leased*. Consequently, when a node leaves or is unplugged from the network, the leases keeping the extension alive fail to be renewed. When this occurs, the instantiated extension is discarded and the extension functionality is dynamically extracted out of the join-points.

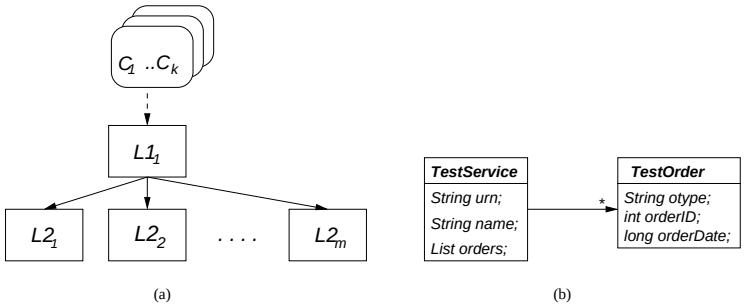


Fig. 6. (a) Three level configuration used for measurements and (b) The local data structure present at each service.

5 Performance results

In this section, we complete and clarify the architectural description by discussing step by step the creation of a spontaneous container with three of the extensions mentioned above (CM, CMP, and ACM); the TM extension has been described and evaluated separately [31]. We include performance measurements to give a clear idea of the costs involved and where optimizations are needed. This container has been implemented as a prototype and it is being extensively used to deploy novel applications over ad-hoc networks.

For the extensions we have used standard libraries (ACM) and developed some parts as needed (CMP and CM). We discuss only the aspects of the extensions to the extent that they are important to understand how they can be embedded within the architecture. However, there are many ways to implement this functionality and the ones we use are just an example of how to go about it.

5.1 Experimental setup

To evaluate the performance we use a varying configuration with three layers of nodes (Figure 6.a). At level zero, nodes act as clients invoking a service implemented at level one. Clients send a request at a time but do not have idle time. As soon as a response arrives, the next request is sent. A number of k clients ($C_1..C_k$) use concurrently a service on level one ($L1_1$). Similarly, the service at level one does a sequence of remote invocations to the services $L2_1..L2_m$ (there are m such services) at level two. Thus, every experimental configuration is characterized by the number m. We use the notation $(1, m)$ to stress the fact that there is one service on level one and m services at level two.

On both levels, each service call performs a number of local operations that update the data structure shown in Figure 6.b. A local operation iterates over the elements in the `orders` list and updates the state of each `TestOrder`. All service invocations across levels are remote calls and all services reside in different nodes.

For the purposes of this paper, we considered all configurations $(1, 1)\ldots(1, 5)$. For each configuration, we vary the number of clients k until the maximal throughput is reached and then measure the response time. The throughput is the average number of invocations per second (inv/sec) performed by all clients. The response time is the average time to complete a call at the client level. For the conciseness of the exposition, we present in this paper the detailed measurements of the response time (as being the most relevant for the desingn of spontaneous container). A more detailed analysis describing the throughput for all measurements and including measurements for more complex configurations (with several nodes on level one) is available [32].

The analysis is a worst case scenario in that the load across all nodes is kept artificially high. The idea is to get measurements that act as lower bounds, since in practice operations are likely to run in disjoint subsets of nodes and therefore be less demanding in terms of the resources because a typical node will not generate as much traffic nor so complex service invocations as those used in the tests.

To evaluate the feasibility of the container we perform two sets of experiments. The first set of experiments considers applications running on a fixed network, where we use Pentium III 600Mhz nodes interconnected using a 100Mbs Ethernet LAN. Since we are interested in architectures for wireless environments, the second set of experiments considers a pure wireless network, where we use Pentium III 400Mhz mobile nodes (lap-tops) interconnected using a 11Mbs wireless LAN (WLAN). The WLAN is configured in ad-hoc mode. We use Java JDK 1.2.2 on each node. We first present the fixed network measurements and then compare them with equivalent configurations in the wireless environment.

5.2 Performance with plain Java services

As a base line for the measurements, we run a series of tests with no extensions involved. Essentially, we are measuring the performance of Java and of making remote calls. Figure 7 gives response time for all configurations.

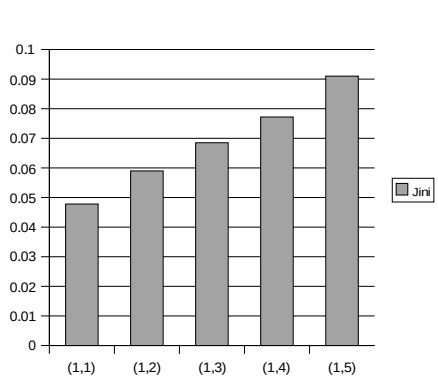


Fig. 7. Response times (s) with no extensions

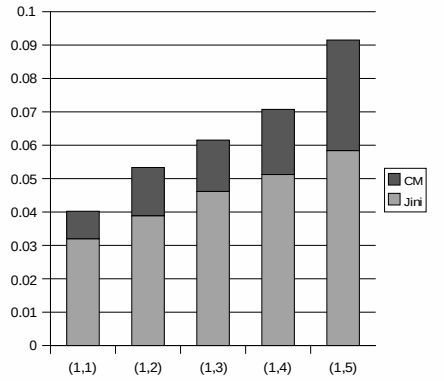


Fig. 8. Response times (s) for the (1,m) measurements, with CM.

5.3 1st Extension: implicit context (CM)

In a distributed system, implicit information must be transparently attached to the parameters of the call at the caller's side and be detached at the callee. Context is transferred in the opposite direction from the callee to the caller together with the return values. Using this mechanism, non-functional information like authentication tokens or transaction identifiers can be transferred between peers.

For this purpose, the extension base distributes the CM extension, which replaces the communication layer of existing services with a new communication layer capable of transferring additional data on the same network connection. The new communication layer checks for every connection whether implicit context must be sent or received from the peer. This functionality does not use the interception mechanism of the PROSE system.

To measure the efficiency of the new communication layer, we distribute CM to all nodes. Then we run the test application when no implicit context data is transferred between peers. The observable performance decrease corresponds to the handshakes incurred by the new communication layer. Figure 8 illustrates the response time of the test system. The total height of the bars represents the response time, in seconds, of all (1,m) configurations. The dark gray part represents the time spent in the new communication layer. With no implicit context data generated by the application, the response time increase is between 0.01 and 0.03 seconds.

Note that the values for Jini response times in Figure 8 (gray segments) do not build upon the values in Figure 7. The reason is that we measure the response time *when the maximal throughput of the system is reached.* E.g., the response times in Figure 7 are measured when the Jini system is loaded with 140 inv/s, while the response times in Figure 8 are obtained when the more complex Jini-CM system reaches its maximal throughput at 90 inv/s. The reason

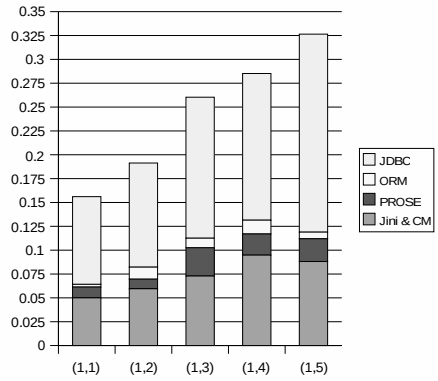


Fig. 9. Response times (s), with the CM and CMP extensions

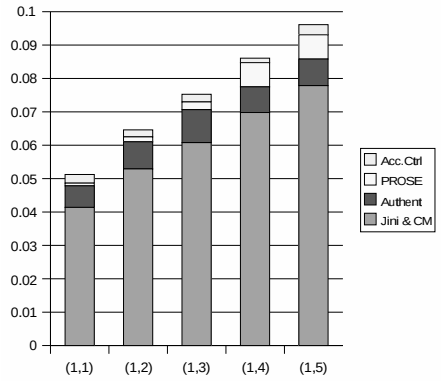


Fig. 10. Response times (s), with the CM and ACM extensions.

behind this measurement methodology is to avoid repeating information between measurement groups, while showing in each case values obtained at a system load which is relevant in practice.

5.4 2nd Extension: implicit context and persistence (CMP)

In the application server area, standards that promote transparent persistence have emerged. A good example is the container managed persistence promoted by EJB [22], but more complex models (e.g., JDO [23]) have also been proposed. Their benefits have been analyzed elsewhere [7, 6].

A spontaneous container could be used similarly for securing the state of any Java-based services. Following this idea, we have developed a solution adapted to the dynamic character of a spontaneous container. It consists of an object-relational mapper (ORM) and relies on capturing object field changes at run-time using PROSE. The component is small (100KBytes) and can save, restore and update the state of entire (Java) object graphs.

The installation of the ORM is performed by the CMP extension. The CMP contains database connectivity parameters and mappings specific to the current network container. After the instantiation, the mapper searches the memory object space of the Jini node. If services, identified by their globally unique Jini IDs [4], are known to the local database, ORM attempts to restore their state. If not, it attempts to store the transitively reachable closure of objects in the local database. Like in commercial container, it is not practical to store the whole object space of an applications. We assume that the applications publish a declarative description on the persistence root-objects.

Finally, the ORM inspects once again the object space and discovers all fields that have to be synchronized with the database. For all these fields, it installs the PROSE extension glue that reports their modification. The specification of

the fields to be watched is generic. As an example, consider an excerpt of a specification from the network meta-data:

```
1 <persistent_service>
2   <package_name>ch.ethz.inf.midas</package_name>
3   <class_name>MyJiniBean</class_name>
4   <primary_key>jiniServiceID</primary_key>
5   <persistent_field>foo.*</persistent_field>
6 </persistent_service>
```

This excerpt specifies on line 6 that all fields whose name matches the regular expression `"foo.*"` and belong to class `MyJiniBean` should be made persistent. It additionally specifies that the primary key of objects of type `MyJiniBean` is `jiniServiceID`.

For the specification of persistence we tried to match existing approaches in the EJB container model. PROSE allows fields to be guarded by means of regular expressions. We use this feature to provide powerful pattern-matching rules for persistence specification.

The response time of the nodes when running with implicit context and container managed persistence added to all services is depicted in Figure 9. Each bar in Figure 9 represents the total response time, in seconds, of the measurements with one L_1 service and several L_2 services. The gray section at the bottom represents the time spent together by the pure Java service and the implicit context functionality. The dark gray section above represents the time spent by PROSE to capture field modifications and submit the corresponding field modification events to the ORM. The white part represents time spent in the ORM to map field updates to database update operations. Finally, the light gray part at the top is the time spent in connections to the database (JDBC).

To measure the average time spent in each section above we incrementally configured the spontaneous container and measured the difference in response times. For the time spent in JDBC calls, e.g., we subtracted the time obtained with a persistence extension from the time we obtained when configuring the extension with a faked database connection of zero response-time.

It is easy to see that total response time increases with greater values for m: the operation is much more complex and it involves communication with the database for each service involved. Most of the response time is due to either the Jini or to the JDBC part.

5.5 3rd Extension: security and context (ACM)

The ACM extension is distributed to each joining node. When inserted, it either creates an ad-hoc identity (key pair) for the node or uses an existing one. The identity is generated using network-specific knowledge. For outgoing connections, the extension authenticates the node against other nodes, while for incoming ones it authenticates the peers. The transfer of authentication data is performed using the functionality provided by the implicit context extension.

The ACM extension is updated by the extension base each time a policy change occurs. It contains information on all known identities and all known services. The access control glue functionality intercepts all remote calls of services on the current node and denies or grants access according to its state. The state of an access control extension is represented by an access control list.

Figure 10 illustrates the response time of the first set of tests when nodes join a container configured to create ad-hoc identities and perform access control for each service call. Each section, from the bottom to the top, represents the time spent for Jini and the CM extension, transferring identities by ACM (dark gray), interception of remote calls by PROSE (white) and access control matrix access (top-most, light gray). As before, the overwhelming part of the time is spent in Jini and the communication layer, and just a minor part is spent in PROSE.

5.6 Throughput for fixed and wireless networks

Our main objective here is to show that the spontaneous container is a feasible technology in both mobile and fixed environments and to evaluate the performance degradation due to the lower bandwidth and CPU speed. Because of hardware restrictions, we have evaluated only the (1,1) and (1,2) configurations. Figure 11 contains a comparison of the fixed network and mobile network spontaneous containers in terms of throughput. For the WLAN/mobile nodes experiments, the performance decrease is not significant, given the limitations in bandwidth and computing power. Thus, over 100 inv/s can be achieved in both configurations on a pure Jini network. Thus, when adding context management (CM), throughput reaches between 51 and 86 inv/s. With container managed persistence (CMP), the wireless spontaneous container enables between 10 and 15 inv/s.

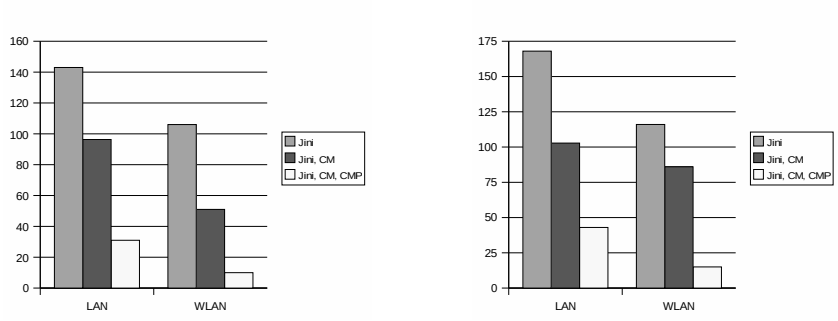
The time spent in each component (CMP,ACM) depends on the number and kind of extensions activated in each node. This shows how the spontaneous architecture addresses the issue of energy consumption. By inserting only the extensions that are needed in a particular environment, it reduces the memory footprint and resource usage of mobile nodes.

6 Discussion and related work

6.1 Related work

At the language level, solutions for adaptation problems range from Meta-Object Protocols [15, 13] to object-based inheritance models [42, 18].

AOP ideas have recently given raise to significant efforts in the area of distributed component models. The idea is to express distributed system properties like load balancing and coordination using aspects [35, 2]. Auto-adaptive systems [3] have recognized the importance of efficiently extending applications to deal with frequent context changes. Also, the limitations implied by static



(a) (b)

Fig. 11. (a) Comparative throughputs (inv/s) between the fixed and mobile infrastructure for the (1,1) configuration and (b) Comparative throughputs (inv/s) for the (1,2) configuration.

container models are underpinned by [12]. Further on, [29] analyses the role of aspect-orientation for expressing infrastructure services and shows some of the deficiencies of the traditional container model.

There has been a significant amount of work on concrete aspects of information systems in, or in connection with, mobile computing environments (e.g., indexing, caching, update consistency, etc). Adaptability in software system architectures for mobile computing is addressed in [11]. Dynamic adaptation of objects to provide non-functional properties like persistence is addressed by incremental object composition [20], while context relations are considered in [37].

Some initial efforts have been made by extending existing middleware platforms. R-ORB [44], for instance, is a context sensitive object request broker based on reconfigurable hardware. In the context of Java, [38] provides a network-wide virtual machine that defines dynamic service components (e.g., monitoring or security). Recent efforts in reflection-based middleware [5] address the very same problem of adaptability. The goal there is to perform QoS adaptations of the CORBA service layer in response to changes in the run-time environment (e.g., network resources) [41]. This work, however, addresses only conventional middleware platforms not the type of ad-hoc networks we are exploring.

6.2 Discussion

In the general case, extensions range from service-generic to fully service-customized. For the generic ones, no specific information on services implementation is needed. For instance, one can replace the communication layer of a service with a new one that encrypts data without any particular knowledge of the adapted service.

Specific adaptations may require a either a black-box view (information on particular interfaces) or glass-box view (knowledge on how a service is implemented) of the service to be adapted. For the spontaneous container to be efficient, each extension base will have many extensions ranging from generic to service-specific, while applications will provide this information if necessary. This approach, also followed by [30], accommodates incremental evolution of the software at every node (because generic extensions will still work even if the node application has changed).

One important issue is how a spontaneous container should address the security of devices entering a network (e.g., to prevent a "malicious" extension base distributing maliciuos infrastructure aspects). Cryptographically signed extensions, supported by PROSE, allow mobile nodes to define a policy policy fiels specifying trusted code bases. In general, a more flexible mechanism is needed. Such a mechanism should blend policy-based security checks and user interaction. Thus, a user should be able to specify what kinds of extensions are trusted by default (based on their signatures), and what extensions should be allowed only with an explicit user acknowledgment.

Unavoidably, mobile and ad-hoc environments face unexpected node detachments. In a spontaneous container, this is the case when a mobile node leaves the container while being actively involved in a distributed computation. Spontaneous containers rely on a RMI-like interaction between clients and servers. With RMI, abrupt node detachment may result in inconsistent distributed computations. To address node detachement, one can enance the container with a transactional extension [?,?]. Thus, one can avoid global inconsistencies by aborting those transactions that involve detached nodes. To address node detachment in a more general fashion, an advanced language and distribution support based on mobile computations [10] could be used.

The spontaneous container approach can be used to optimize information systems so that they can be efficiently used in an ad-hoc computing environment. In this regard, the spontaneous container is public domain software and, as our experiments have shown, can be a very powerful platform for experimenting with issues related to mobility and reconfigurability of the IT infrastructure.

Even as a first step, the results provided indicate that spontaneous adaptation of services is possible in both fixed and wireless environments. The experiments performed are really a stress test. For a distributed application, interactions are very complex (involving between 6 and 30 remote accesses) and, once persistence is involved, also very costly (every service triggers a remote transaction in the centralized database). A more realistic load, specially if it is manually generated by users, will be much less demanding and results in a considerable higher throughput. The network bandwidth plays an important role here specially given that it is quite limited in existing wireless environments. This bandwidth, however, will only grow in the future and advances in network protocols such as dynamic scatternets, multi-hop frequency access, and simply larger bandwidth will alleviate this situation.

With this in mind, the results provided are quite encouraging. We do not pretend that the container is ready to be used in a large scale network. Nevertheless, the experiments show that the mechanisms for creating a spontaneous information system do not have a significant effect in the overall performance when compared with the intrinsic cost of container managed persistence or transactions. These high costs are inherent to the nature of the extensions and have little to do with the method used to insert them into the application. Even if the insertion happens at deployment time, the experiments show that the larger part of the costs are produced at run-time and are independent of the insertion method chosen.

7 Conclusion

In this paper we have presented a system that allows computing nodes to exchange functionality and build information systems in a dynamic manner. The term we use is that of a spontaneous container. A spontaneous container dynamically adapts nodes using aspect-oriented extensions. We have presented extensions for security, transactions, and container-managed persistence. Applications of spontaneous containers range from entirely ad-hoc service communities to infrastructure-centric service adaptations. In the ad-hoc case, extensions are exchanged between nodes on a peer-to-peer basis. In the infrastructure centric case, base stations associated to, e.g., a fair ground, distribute extensions modeling the organization's policy. A spontaneous container allows organization-wide middleware properties, and service adding or removal without interrupting operations. The measurements made with the prototype, both for a fixed and a mobile network infrastructure are quite encouraging. They illustrate that the adaptation mechanism we present can be used to effectively transform a service community into a spontaneous container. The real advantage of this system is the flexibility of the architecture that allows the exploration of new forms of interaction in mobile and wireless networks.

8 Acknowledgements

The work presented in this paper was supported (in part) by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322. We are grateful to Intel for a generous equipment grant that was used to build part of the experimental platform used in the paper. The spontaneous container prototype and its sub-components are freely available as open source modules downloadable from `http://prose.ethz.ch`.

References

1. M. Aksit, K. Wakita, J. Bosch, and L. Bergmans. Abstracting Object Interactions Using Composition Filters. *Lecture Notes in Computer Science*, 791:152, 1994.

2. M.P. Alarcon, L. Fuentes, M. Fayad, and J.M. Troya. Separation of Coordination in a Dynamic Aspect Oriented Framework. In *1st Intl. Conf. on AOSD, Enschede, The Netherlands*, April 2002.
3. L. Andrade and J.L. Fiadeiro. An architectural approach to auto-adaptive systems. In *Proceedings of the 22nd ICDCS Workshops*, pages 439–444, Vienna, Austria, July 2002.
4. K. Arnold, A. Wollrath, B. O'Sullivan, R. Scheifler, and J. Waldo. *The Jini Specification*. Addison-Wesley, Reading, MA, USA, 1999.
5. D. Arregui, F. Pacull, and J. Willamowski. Rule-Based Transactional Object Migration over a Reflective Middleware. In *Middleware 2001*, volume 2218 of *LNCS*, pages 179–196, 2001.
6. M. P. Atkinson, L. Daynes, M. J. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java. *SIGMOD Record*, 25(4):68–75, 1996.
7. M. P. Atkinson and R. Morrison. Orthogonally Persistent Object Systems. *VLDB Journal*, 4(3):319–401, 1995.
8. J. Baker and W. Hsieh. Runtime Aspect Weaving Through Metaprogramming. In *1st AOSD, Enschede, The Netherlands*, pages 86–95, April 2002.
9. L. Bergmans and M. Aksit. Composing Crosscutting Concerns using Composition Filters. *CACM, Special Issue on Aspect-Oriented Programming*, 44(10):51 – 57, October 2001.
10. L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, January 1995.
11. E. de Lara, D.S. Wallach, and W. Zwaenepoel. Puppeteer: Component-based Adaptation for Mobile Computing. In *USITS-01*, Berkeley, CA, March 2001.
12. F.Duclos, J. Estubiler, and P. Morat. Describing and Using Non Functional Aspects in Component Based Applications. In *AOSD, Enschede, The Netherlands*, April 2002.
13. J. Itoh, Y. Yokote, and R. Lea. Using Meta-Objects to Support Optimisation in the Apertos Operating System. In *Proc. of COOTS*, Berkeley, USA, June 1995.
14. N. Kassem. Designing Enterprise Applications with the Java 2 Plaform. Sun J2EE Blueprints, http://java.sun.com/j2ee/download.html, June 2000.
15. G. Kiczales and J. des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.
16. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP '97 ,Jyväskylä, Finland*, volume 1241, pages 220–242. Springer-Verlag, New York, NY, 1997.
17. J. Kleinoeder and M. Golm. MetaJava — A Platform for Adaptable Operating-System Mechanisms. In *ECOOP'97 Workshop Reader*, LNCS. Springer, 1997.
18. G. Kniesel. Type-Safe Delegation for Dynamic Component Adaptation. *LNCS*, 1543:136–137, 1998.
19. T. J. Lehman, A. Cozzi, Y. Xiong, J. Gottschalk, V . Vasudevan, S. Landis, P. Davis, Bruce K., and P. Bowman. Hitting the distributed computing sweet spot with TSpaces. *Computer Networks (Amsterdam, Netherlands: 1999)*, 35(4):457–472, March 2001.
20. M. Mezini. Dynamic Object Evolution without Name Collisions. In *ECOOP'97*, volume 1241 of *LNCS*, pages 190–219. Springer, 1997.
21. Microsoft Corporation. The .NET Internet page. Accessible on-line at: http://www.microsoft.com/net/, 2001.
22. Sun Microsystems. Enterprise Java Beans Specification, Version 2.0, August 2001.
23. Sun Microsystems. Java Data Objects (JSR 12), May 2001.

24. H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and F. Kimura. OpenJIT : An Open-Ended, Reflective JIT Compiler Framework for Java. In *Proc. of ECOOP'2000, Springer Verlag*, 2000.
25. A. Oliva and L.E. Buzato. The design and implementation of Guaraná. In *Proc. of the 5th USENIX COOTS*, pages 203–216. The USENIX Association, 1999.
26. D. Orleans and K. Lieberherr. DJ: Dynamic Adaptive Programming in Java. In *Reflection 2001*, Kyoto, Japan, September 2001. Springer Verlag.
27. K. Ostermann. Dynamically Composable Collaborations with Delegation Layers. In *Proc. of ECOOP'2002)*, Malaga, Spain, 2002. Springer.
28. R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In *Reflection 2001*, pages 1–24, Kyoto, Japan, September 2001. Springer Verlag.
29. R. Pichler, K. Ostermann, and M. Mezini. On Aspectualizing Component Models. *Software Practice and Experience*, 2003.
30. S. R. Ponnekanti, B. Lee, A. Fox, P. Hanrahan, and T. Winograd. ICrafter: A Service Framework for Ubiquitous Computing Environments. *LNCS*, 2201, 2001.
31. A. Popovici and G. Alonso. Ad-Hoc Transactions for Mobile Sevices. In *Proc. of the 3rd Intl. TES/VLDB Workshop*, Hong Kong, China, August 2002.
32. A. Popovici, G. Alonso, and T. Gross. Design and evaluation of spontaneous container services. Technical report no. 368, CS Department, ETH Zurich, 2002.
33. A. Popovici, G. Alonso, and T. Gross. Just in time aspects: Efficient dynamic weaving for java. In *AOSD03, Boston, USA*, March 2003.
34. A. Popovici, T. Gross, and G. Alonso. Dynamic Weaving for Aspect Oriented Programming. In *AOSD02, Enschede, The Netherlands*, April 2002.
35. E. Putrycz and G. Bernard. Using Aspect Oriented Programming to build a portable load balancing service. In *22nd ICDCS*, Vienna, Austria, July 2002.
36. B. Redmond and V. Cahill. Towards a Dynamic and Efficient Reflective Architecture for Java. In *Workshop at ECOOP 2000*, Cannes, France, June 2000.
37. Linda M. Seiter, Jens Palsberg, and Karl J. Lieberherr. Evolution of object behavior using context relations. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Software Engineering Notes, pages 46–57, New York, October 16–18 1996. ACM Press.
38. E.G. Sirer, R. Grimm, A.J. Gregory, and B.N. Bershad. Design and Implementation of a Distributed Virtual Machine for Networked Computers. In *Symposium on Operating Systems Principles*, pages 202–216, 1999.
39. P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N Degrees of Separation: Multi-dimensional Separation of Concerns. In *1999 ICSE*, Los Angeles, CA, USA, 1999.
40. The Object Management Group. The CORBA Component Model RFP. Available at http://www.omg.org/docs/orbos/97-05-22.pdf, 1997.
41. E. Truyen, B. Joergensen, and W. Joosen. Customization of Component-Based Object Request Brokers through Dynamic Configuration. In *TOOLS Europe'2000, IEEE Press*, pages 181–194, June 2000.
42. D. Ungar and R. B. Smith. Self: The Power of Simplicity. In *Proc. of OOPSLA*, pages 227–242, 1987.
43. Xerox Corporation. The AspectJ Programming Guide. Online Documentation, 2002. http://www.aspectj.org/.
44. S. S. Yau and F. Karim. Reconfigurable Context-Sensitive Middleware for ADS Applications in Mobile Ad-Hoc Network Environments. In *Proc. of the 5th ISADS.*