Design and Evaluation of Spontaneous Container Services
Technical Report Number 368

A. Popovici, G. Alonso
Department of Computer Science
Swiss Federal Institute of Technology (ETHZ)
ETH Zentrum, CH-8092 Zürich, Switzerland

Abstract
Technologies like Jini offer spontaneous discovery and utilization of services. Container technology (e.g., Enterprise Java Beans) allows the transparent adaptation of the application logic at deployment time. These two approaches tackle different sides of the same problem: how to cope in a flexible manner with dynamic changes in the environment where the application runs. Combining the two approaches leads to a virtual container service infrastructure in which services or functionality extensions can be dynamically added to or removed from an application as needed. This is particular important nowadays given the increasing pervasiveness of wireless and ad-hoc networks as well as peer-to-peer interaction. In this paper we discuss how this can be done in the context of Java. The system we present effectively and efficiently transforms a Java spontaneous network into a virtual container for services and extensions, thereby providing the advantages of both approaches in a single platform. As a test case, we use this infrastructure to implement important data management functionality: transactional interaction, access control, and container managed persistence, and show how it can be used in an ad-hoc network environment.

1 Introduction
Networking environments that either avoid a fixed infrastructure or allow direct, spontaneous interactions between peers are rapidly becoming available for practical use. To support these new computing environments, it is crucial to abandon the current paradigm in which software capabilities are determined at development time. To a certain extent, this change is already happening. For instance, modern container models [Mic01a] separate the business logic 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 be implemented as part of every application. Similarly, the increasingly widespread use of wireless networks is leading to another form of adaptation: dynamic service discovery of the type found in systems like Jini [AWO+99].

Unfortunately, the hardware and network platforms underlying future information systems are evolving much faster than the software infrastructure. New computing environments such as spontaneous or ad-hoc networks and peer-to-peer interaction challenge even the flexibility provided by container and dynamic service discovery models. In these new environments, nodes can independently join and leave a community of services. Nodes do not know in advance which other nodes they will interact. One cannot assume a pre-defined environment, e.g., the Internet, as nodes build their own network on the spot. This makes it very difficult to rely on existing hardware or software infrastructure. As a response to this limitations, ambitious middleware platforms, which promote adaptation, have emerged. R-ORB [YK01] is a context sensitive object request broker based on reconfigurable hardware. In the context of Java, [SGGB99] aims at a network-wide virtual machine that defines dynamic service components (e.g., monitoring or security).

In spite of these efforts, the appropriate information system support for such new computing environments is largely missing. A flexible software infrastructure is needed, in which functionality should be factored out of an application to become a dynamic property of the computing environment. By “computing environment”, we mean any set of two or more applications that decide to interact with each other and might build their own communication network for that purpose. When an application joins a new computing environment, its functionality should be extended and/or adapted as needed. The novelty of this idea is to see the computing environment as the provider of such extensions (rather than, e.g., a static container). Thus, the nodes that make up this ad-hoc computing environment should be able to spontaneously generate any service infrastructure they might need.

In this paper we present a Java-based service infrastructure capable of run-time service adaptation. The system we have built effectively transforms a Jini
network into a dynamic, secure service environment functionally equivalent to an Enterprise Java Beans (EJB) container providing container managed persistence, access control and transactional interaction. We concentrate on this functionality because it plays a crucial role in any information system and provides an excellent test-bed to better understand the problems associated with these new computing environments. Using this infrastructure, nodes can dynamically acquire extensions that make their state persistent (the state being stored at a base station or at another node), their interactions transactional (with arbitrary levels of nesting and interacting with either base stations or directly among them), and subject them to an access control policy (to access information in other nodes or at base stations).

These extensions allow the formation of ad-hoc information systems in an entirely spontaneous manner. The resulting architecture has many advantages. It allows the interoperability of mobile and fixed network applications. The container model is an accepted and well understood technology. We believe that this is a significant advantage, because it does not require a particular hardware setting [YK01] or a network-wide virtual machine [SGGB99]. By combining discovery and service adaptation, the approach becomes applicable in a wide range of scenarios from the conventional (e.g., application deployment or code upgrades) to the most advanced (e.g., ad-hoc networks or peer-to-peer interactions over wireless networks).

In the paper, we first motivate the work by providing an example scenario that extends existing commercial systems (Section 2). Then we describe the architecture and how functionality can be added at run-time to applications running on a JVM (Section 3). Based on this architecture, we have implemented a spontaneous container providing container managed persistence, access control and transactional interaction. We explain how the container works by discussing every necessary extension step by step (Section 4). We have developed extensions that demonstrate the potential of the approach and what is involved in building a spontaneous information system. 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 and conclude the paper in Section 5.

2 Motivation

2.1 Example scenario

Imagine a conference, trade show, meeting, or exhibition hall where participants are provided with computing devices such as lap-tops or PDAs (we will refer to these devices from now on as nodes)\(^1\). Assume there is a common infrastructure that allows service publishing and discovery (e.g., Jini running on a combination of fixed and wireless network). Nodes can communicate with either a base station or directly with each other. Using this ad-hoc platform as the basis, 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, without relying in any centralized infrastructure. That is, nodes should be able to exchange among themselves whatever functionality extensions they need and, after that, be capable of interacting transactionally among them, have their state persistently stored somewhere (other nodes or a base station), and follow a simple access control policy.

Important requirements are:

(a) The nodes carry with them only the basic platform support for run-time adaptation,
(b) the functionality extensions can be provided by any other node including, but not limited to, base stations,
(c) functionality extensions are only leased; if a node leaves the computing environment the extension disappears, and
(d) functionality extensions can be dynamically updated as policy changes occur without having to interrupt service or having to modify the nodes.

2.2 Target architecture

One way to implement such a scenario is to treat each node (i.e., each service made available by a node) as a component that can be extended at run-time with the necessary functionality. The functionality extensions can be obtained either from a base station or from other nodes upon joining the computing environment. Later on, the node might acquire additional extensions or distribute extensions to other nodes as needed. Upon leaving the network, all extensions are removed. Examples of such extensions are:

A context extension that modifies the communication layer so that context information is added to service calls. This context information will then be used as the basis for access control and transactional interaction.

A persistence extension that maps the local state of a service (e.g., an order received by a merchant) to stable storage located into some other node or base station. This same extension restores the state of the service after a crash.

\(^1\) A similar scenario focusing on basic communication services such as messaging and person identification is already being commercially exploited with over a thousand nodes being connected: http://www.shockfish.com/.
A transactional extension enforces transactional correctness. This extension installs a mini transactional monitor in each node so that invocations to services are trapped and made transactional.

Additional extensions dealing with authentication, access control, or billing are added as needed depending on location, circumstances, number of participants, etc.

2.3 Related work

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). To our knowledge, much less has been done in the area of software system architecture and infrastructures for information systems in these environments. Nevertheless, some products are starting to appear although still without supporting run-time adaptation. For instance, there are products that export the functionality available in conventional application servers as Jini services [Teo02] so that this functionality can be accessed by mobile devices. However, this is just a matter of interface adaptation and not the run-time adaptation we are proposing.

The work most relevant to what we propose are recent results in software engineering, namely on Aspect Oriented Programming (AOP) [KLM+97]. The basic idea of AOP is to abstract out of an application all orthogonal concerns (functionality) so that they can be treated separately [TOHS99]. 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 [XC01, LK98], for instance, these crosscuts could be the invocations of some method(s) of a set of classes.

There exists as well some initial work on adaptive middleware based on AOP and reflection [CBPC01, APW01]. The idea there is to perform QoS adaptions of the CORBA service layer in response to changes in the run-time environment (e.g., network resources) [TJJ00]. This work, however, addresses only conventional middleware platforms not the type of ad hoc networks we are exploring.

Finally, run-time changes to a program are usually performed in languages that explicitly support run-time adaptability such as composition filters [AWBB94] or reflection [KdR91, IYL95, KG97, OSM+00]. More recently, AOP ideas have been proposed and implemented as part of the PROSE (PROgrammable extensiOns of sErVices) system [PGA02]. PROSE works on a modified Java Virtual Machine (JVM) and allows the functionality of an application to be extended at run-time. The architecture discussed in this paper has been built on top of PROSE. However, the same ideas can be implemented with any other system that supports run-time adaptability.

3 Spontaneous container architecture

In this section we first briefly discuss PROSE, the language and run-time system used to develop the spontaneous container architecture. We then show how to use PROSE to perform container-like adaptation so that applications can be extended with the necessary functionality. Finally, we show how to extend every node so that it can exchange, provide, and receive extensions at run-time, thereby providing the basic mechanism for spontaneous interaction.

3.1 Language and run-time support

For reasons of space and scope, we will not discuss PROSE in detail. For the purposes of this paper, PROSE is just a run-time adaptation system. The interested reader can find more information about it recent publications [PGA02, PGA01]. Here we provide a brief overview of its basic mechanisms so that the programming context can be understood.

PROSE is based on a modified JVM that can perform interception and weaving at run-time. However, an application will not see any difference with a standard JVM. The PROSE JVM provides an API for inserting and removing extensions. These extensions specify the code for the extension (what to do) and where it needs to be called (when to do it). Typical examples are:

- Invoke the extension `txBegin` before entering any method whose name matches the regular expression `".\*tx\.*"` or involves a remote method invocation (RMI).

- Invoke the extension `storeNewValue` whenever a field is modified in any class whose name matches the regular expression `".\*EntityBean\.*"`

PROSE extensions are regular Java objects that can be sent over the network. Signatures are used to guarantee their integrity. PROSE is independent of the programming language: it concentrates on the execution inside the JVM and not on the source compiled to produce the byte-code. 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.
Requiring all applications to run on Java is currently a limitation. This constraint becomes less critical as similar technology spreads (e.g., the .NET platform). The same type of interception could be done in other platforms (namely, in .NET). In fact, with the appropriate compiler support, aspect-oriented runtime changes can be introduced in any application.

In the architecture we distinguish between two types of adaptations. These adaptations use the same PROSE mechanisms but they play different roles. The first type is the extension functionality. The second type 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.

3.2 Adapting functionality through extensions

Consider three nodes, A, B, and C that want to interact as follows. Node A initiates a distributed computation within method \( m_A \) (Figure 1.a). It first performs some local operations and then invokes method \( m_B \) of the 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 “*”. \( m_B \) 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 1.b shows the control flow once the extensions are in place. As before, the computation starts in \( m_A \). When the remote invocation to \( m_B \) is initiated, the TM extension glue traps the call (step 1, dark gray). The extension 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 (double hatched), located in the communication layer, marshals the implicit context data together with the parameters of the call (step 2), before the invocation takes place (step 3).

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 (simple hatched) to check whether A has the right to access \( m_B \) (step 4). If access is granted, the TM extension of B (dark gray) 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 (asterisk). The CMP extension glue intercepts the state change and notifies the object-relation mapper (step 6) that will perform the corresponding update in the corresponding 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, double hatched) and passed to the TM extension for processing (step 10). After that, execution of \( m_A \) resumes with analogous steps for the call to \( m_C \).
This scenario illustrates the adaptations needed to transform a single node of a spontaneous network into a mini container. If each node is extended in a similar manner, all services will behave as if they would have been deployed in a virtual container.

3.3 Creation of extensions

Extensions can be written knowing or not knowing the source code they will extend. 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 one 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. In this case, extensions are for software maintenance and upgrade purposes.

The adaptation mechanism described above supports any of these three type of extensions. For reasons of space, however, we cannot discuss the software engineering issues involved and when to use each type of extension. It suffices to explain how such extensions can be created in the context of a spontaneous container. This has been done by extending PROSE with support for the equivalent of the deployment descriptors used in EJB architectures [Mic01a]. This support comes in the form of a meta-data repository (network meta-data in Figure 2.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 extension functionality and glue from these specifications. The resulting PROSE extension object [PGA02] can be sent through the network.

3.4 Management of extensions

Our approach is fully interoperable with Jini. In addition, all nodes can send extensions to other nodes and can act as lookup services (LUS), if necessary. For simplicity in the exposition, however, we will assume the extensions are being sent from one base station (the same procedure can be performed by any node) and that a lookup service is available.

Figure 2a illustrates the basic mechanism used to discover and use new services in Jini. A service (B) joins the computing environment by registering a proxy pB at a nearby lookup service (step 1). Other nodes can query the LUS (step 2), obtain pB and use the service B (step 3). Alternatively, nodes can ask to be notified when a service matching a certain template joins or leaves the Jini community.

Our spontaneous container architecture extends this basic idea as follows. We define an adaptation service, based on PROSE, that allows the uploading of extension objects into a node. Each node must carry an adaptation service, which is accessible as a regular Jini service. Figure 2b illustrates service B running together with an adaptation service (marked in the figure with PROSE) on the same JVM. Like
any other service, the adaptation service joins the Jini 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). The immediate effect is for the extensions functionality to be instantiated at B (step 4') and the monitoring of the corresponding joint-points activated (4")

Using the leasing mechanism provided by Jini [AWO+99], the extension object sent to each node is 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 glue functionality is dynamically extracted out of the join-points.

4 Spontaneous container evaluation

In this section, we complete and clarify the architectural description by discussing step by step the creation of a spontaneous container with the four extensions mentioned above (CM, TM, CMP, and ACM). We also 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), developed some parts as needed (CMP and CM), and used a commercial product (TM). We discuss only the aspects of the extensions to the extent that 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.

4.1 Experimental setup

To evaluate the performance we use a varying configuration with three layers of nodes (Figure 3a). At level zero, nodes act as clients invoking the services 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 (C1−Ck) use concurrently the same service on level one (L1i). Similarly, the services at level one call services at level two. Each service L1i (there are n such services) does a sequence of remote invocations to the services L21−L2m (there are m such services) at level two. On both levels, each service call performs a number of local operations that update the data structure shown in Figure 3b. A local operation iterates over the elements in the orders list and updates the state of each TestOrder.

Every experimental configuration is characterized by the tuple (n, m). For the purposes of this paper, we considered all configurations (1, 1) . . . (5, 5). For each configuration, we varied the number of clients k until the maximal throughput is reached and then measured the throughput and 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 recorded to complete a call at the client level. With this, 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 they need.

To avoid changes of bandwidth, connectivity and availability, we perform all experiments using a cluster of PCs on a local area network (Table 4.1). All service invocations across levels are remote calls and all services reside in different nodes. The nodes are more powerful than high end PDAs but equivalent to lap-tops. During the experiments, the systems were not loaded with user activity. The network provides more bandwidth than wireless networks. The reason to use a LAN is that the type of stress test we conduct cannot be done in current wireless networks due to their low capacity. In a wireless environment, the reduced network bandwidth will be compensated by the fact that a node will generate neither nearly as much traffic nor so complex service invocations as those used in the tests.
4.2 Performance with plain Jini

As a base line for the measurements, we run a series of tests with no extensions involved. Essentially, we are measuring the overhead of Jini and of making remote calls. Figure 4.a illustrates the throughput and Figure 4.b the response time for all configurations. The throughput for the configurations (1, m) is smaller than for the other configurations (2, m) ... (5, m) indicating that a single node on level one is a performance bottleneck in the test performed.

4.3 1st Extension: implicit context

In any 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 the handshakes incurred by the new communication layer. Figure 5.a illustrates the throughput of the test system. In all cases, the total number of invocations per second is down by roughly one third compared to Figure 4. In Figure 5.b, the total height of the bars represents the response time, in seconds, of all (1, m) configurations (corresponding to the leftmost group of measurements in Figure 5.a). 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 increases by approximately 0.03 seconds.

4.4 2nd Extension: implicit context and persistence

In the application server area, standards that promote transparent persistence have emerged. A good example is the container managed persistence promoted by EJB [Mic01a], but more complex models (e.g., JDO [Mic01b]) have been proposed. Their benefits have been analyzed elsewhere [AM95, ADJ+96].

For securing the state of Jini services, a similar approach would be beneficial. Following this idea, we have developed a solution adapted to the dynamic character of Jini. 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.

1. `<persistent_service>`
2. `<package_name>` ch.ethz.inf.midas
3. `<class_name>` MyJiniBean
4. `<primary_key>` jiniServiceID
5. `<primary_field>` name
6. `<primary_field>` foo.*
7. `<primary_service>`

Figure 6: Network meta-data specifying container managed persistence for MyJiniBean.

The installation of the ORM is performed by the CMP extension. The CMP extension inserted into the weaver 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 [AWO+99], are known to the local database, ORM attempts to restore their state. If not, it attempts to store the transitorily reachable closure of objects in the local database.

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, Figure 6 is an excerpt of a specification from the network meta-data. It 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 EJB. 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 throughput and response time of the nodes when running with implicit context and container managed persistence added to all services are depicted in Figure 7. Here, too, the throughput decreases with greater values for m: the operation is much more complex and it involves communication with the database for each service involved.

Each bar in Figure 7.b represents the total response time, in seconds, of the first group of measurements in Figure 7.a. The gray section at the bottom represents the time spent together by Jini 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).

4.5 3rd Extension: security and context

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 8.a illustrates the throughput of the test application when joining a container configured to create ad-hoc identities and perform access control for each service call. The throughput is slightly smaller than in the Jini plus CM extension. Figure 8.b illustrates response times for the (1,m) configurations. 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).

4.6 4th Extension: transactions, persistence, and context

In a spontaneous network, we want to allow transactions of arbitrary complexity. The objective is that, once all nodes have the necessary software layer added to them, the interactions should become transactional. One of the challenges of this scenario is to be able to guarantee transactional correctness without a centralized component. Recently, a solution to this problem has been developed as part of the CheeTah system [PA00] and is commercially available [At00].

CheeTah is essentially a small TP-Monitor that resides in each node of the composite system. CheeTah treats each remote call as a subtransaction of a global root transaction. Thus, a service designed to use CheeTah must wrap the application logic with invocations to the mini TP-Monitor inside each node. The management of the nested transactions is transparent to the application code, and (this is the relevant part) entirely local to the node. No matter how nodes interact with each other, as long as each one of them uses CheeTah, the overall result is correct transactional executions that can also be automatically recovered.

One of the main advantages of CheeTah is that it is very light-weight, less than 300 KBytes of code, and very flexible. Being written in Java, CheeTah is also portable. For our purposes here, CheeTah is the ideal transactional component, since it is portable and small enough to allow transferring the whole code from node to node.

4.6.1 Transactional adaptation

The transactional interaction is implemented by the glue functionality bracketing remote calls. The glue functionality determines whether a remote call corresponds to a root transaction or a sub-transaction. With this information, CheeTah can take over and control the execution of the remote calls as if they were nested transactions.

As an example, consider a node $N_1$ that calls method $m$ of a remote node $N_2$. By intercepting the call, the extension functionality can check whether it is associated with any transaction. If it is not the case, then it associates this call with a root transaction $t_1$. As part of the same call to $N_2$, the implicit context functionality sends the root identifier ($t_1$) and node identifier ($N_1$) so that $N_2$ notices (i) that it is running
a sub-transaction and (ii) the location of the parent.

At $N_2$, the invocation of $m$ first executes the extension glue that deals with implicit context. This functionality extracts (and removes) the hidden parameters and sees a root transaction identifier. Accordingly, $N_2$ starts a local transaction $t_2$ as a sub-transaction of $t_1$. The sub-transaction $t_2$ is associated to the thread where the invocation of $m$ runs. In this way, all remote calls made during the execution of $m$ can be intercepted and treated as sub-transactions of $t_2$. For these calls, $N_2$ associates the root identifier ($t_1$) with its own node identifier ($N_2$). This is all the information a CheeTah TM needs to detect the structure of a nested transaction [PA00].

When all calls complete, CheeTah must be activated again to gather information regarding all sub-invocations of a thread. This information is used for atomic commitment purposes and it includes the number of sub-transactions invoked [PA00]. As in the forward phase, the CheeTah monitors on each side exchange information among themselves using the implicit context feature. When a CheeTah extension sees that a call that just completed corresponds to a root transaction, this extension starts the 2-Phase-Commit (2PC) protocol. The commitment protocol is performed among the extensions and does not involve intercepting calls. In the current form, the system cannot deal with nodes leaving the community during the commit phase. This and similar issues are typical of transactional interaction and beyond the scope of this paper. Our goal here is to show that, within the limits of conventional transactions, transactional behavior can be dynamically added in a spontaneous network.

4.6.2 Transactional specification

```xml
1  <tx_mgr_config>
2      <db_ressource_type>DB_XA</db_ressource_type>
3      <db_pool_size>16</db_pool_size>
4      <obj_db_mapping>PersMgrImpl</obj_db_mapping>
5      <tx_mgr_timeout>30</tx_mgr_timeout>
6    </tx_mgr_config>
7  <transactional_services>
8      <transactional_service>
9          <package_name>ch.ethz.midas</package_name>
10         <class_name>JiniBeanInterface</class_name>
11     </transactional_service>
12 </transactional_services>
```

Figure 9: Network meta-data specifying transactional service for methods defined in JiniBeanInterface.

The transactional specification matches closely the design of the system. On the one hand, corresponding to the TM component, a number of network-wide parameters define the type of transactions to be employed in each node (Figure 9, lines 1-6). Examples of such parameters are timeout specification for in-doubt transactions, transaction categories to be used (local,XA, compensating), semantic of high-level locks, and other configuration attributes of CheeTah [Par00]. On the other hand, corresponding to the join-points and associated glue functionality, the names and types of services to be made transactional must be specified (lines 8 - 11). Specifically, line 11 is used to create a service extension that adds transactional glue code to all methods declared in JiniBeanInterface, irrespective of how they are implemented.
4.6.3 Transactional performance

Inserting the TM extension leads to a visible loss of performance. Figure 10a illustrates the throughput of the system when all service invocations are treated as transactions. The total response time increases significantly (Figure 10b) because for each level one service invocation, now a root transaction, an additional round of 2PC must be carried out. In Figure 10b, the top section of each bar represents the time spent for transactional coordination. The dark gray section in the middle represents the time spent in the TM glue, while the bottom part is the time spent for transparent persistence and application logic.

5 Discussion and conclusions

The low throughput reached when several extensions are inserted, TM in particular, might seem disappointing. However, the experiments performed are really a stress test. Interactions are very complex (involving between 6 and 30 remote accesses) and, once transactions or persistence are involved, also very costly (in the case of TM the 2PC at the end involves many nodes and, in the case of persistence, 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. Obviously the network bandwidth plays a very 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. There are many other data management extensions that can be of great use in such environments that will not incur in such high cost. Examples are extensions for load balancing access to databases in base stations, for replication of RMI calls to a server and a backup, for best-effort logging based on files rather than database transactions, etc. For these extensions, the spontaneous container offers significant advantages over existing infrastructures.

Compared to other platforms that promote adaptability based on specialized hardware or software platforms, our approach is entirely compatible with Jini (the adaptation is exported as as service, the weaver). Even as a first step, the results provided constitute an excellent indication of where 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 power-
ful platform for experimenting with issues related to mobility and reconfigurability of the IT infrastructure.

References

75, 1996.


29, 1995, Monterey, California, USA, pages 147-
158, Berkeley, CA, USA, June 1995. USENIX.


