Ad-Hoc Transactions for Mobile Services

Andrei Popovici and Gustavo Alonso

Swiss Federal Institute of Technlogy Zürich, Switzlerland
{popovici,alonso}@inf.ethz.ch

Abstract. New developments in battery technology, networking, and devices allow the creation of new business models based on mobile computing and not requiring any fixed infrastructure. Mobile electronic commerce is today limited in size but future mobile networks will grow dramatically. In such environments, the participating nodes must be self-organized to collaboratively implement all the services a fixed network would provide. In this paper we present the design and implementation of a system that supports the development of adaptive electronic services. We focus our attention on the problem of transactional interaction between nodes, which is an essential requirement in electronic commerce. To support, this feature, our system allows us to dynamically incorporate the transactional support in mobile nodes and provides the foundation for a self-organizing transaction system. The paper discusses then how groups of collaborating nodes can execute electronic transactions on an infrastructure-less, ad-hoc environment. We conclude with a preliminary performance evaluation.

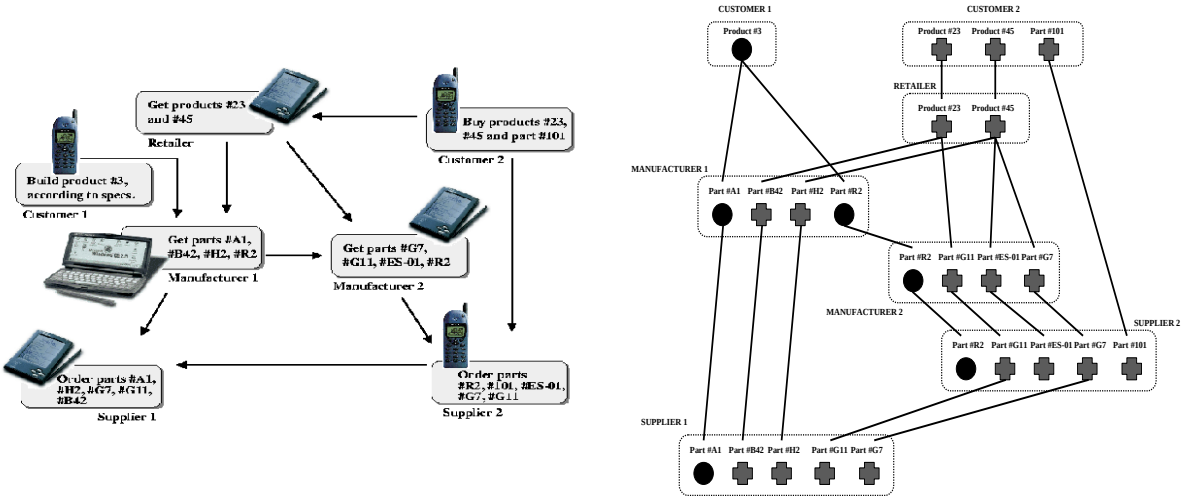
1 Introduction

The widespread use of computing and communication devices is increasingly challenging the way services are provided to consumers. Traditionally, most electronic services have been built around a fixed infrastructure. However, the emergence of handheld and ubiquitous computing renders services that rely on a fixed network infrastructure inadequate in many scenarios. This new type of environments are require the participating nodes to collaborate to implement the functionality a fixed network would provide. They also exhibit the ability to *adapt* the behavior of the participating nodes to the current context (e.g., nearby services, physical location, oranizational unit, etc.).

The design of the infrastructure for electronic services is clearly lagging behind in addressing these new capabilities and constraints. Most existing systems depend on a fixed infrastructure. This dependency is very strong, both at the design and the conceptual level. For instance, all existing solutions for transactional interaction are based on a centralized component [5]. Thus, current products cannot work in a server-less environment where nodes may dynamically move from one application domain to another [15].

As a first step towards eliminating this limitation, we have designed a system capable of providing transactional support to groups of self organizing nodes. The challenge is to provide transactional functionality without having to rely on any infrastructure and while allowing the nodes to change their transactional behavior depending on the group they join. The first requirement, avoiding a fixed infrastructure, implies that the nodes need to be able to organize themselves as a TP-Monitor. This objective can be achieved by making each node an entirely autonomous mini-TP-Monitor. The second requirement, the ability to adapt to different environments, implies that the transactional functionality cannot be hardwired into the node. What is needed is the ability to, at runtime, attach or detach to a node the mini-TP-Monitor functionality.

In this paper, we present a system capable of doing this. In our system, all nodes are members of a spontaneous network (e.g., Jini [2]). This addresses the problem of service discovery and brokerage in a server-less environment. When several nodes get together, one node makes available to the other nodes a small piece of code (less than 300 KBytes) that contains all the necessary transaction management logic. This code is based on the CheeTah system [15]. This solves the problem not having a fixed infrastructure for transactional processing. To provide adaptability, we use the PROSE system [17]. PROSE allows us to systematically modify the behavior of a



(a) (b)

Fig. 1. (a) Example of ad-hoc transactional interaction, (b) its transactional structure.

running application by dynamically combining the (network-specific) transaction processing with the original application logic inside a Java Virtual Machine. We have taken advantage of this feature to redirect incoming and outgoing remote calls to a node through the TP-Monitor code so that the calls can be made transactional in an automatic and transparent manner. This same mechanism could also be used to support functionality like security or network awareness.

The paper is organized as follows: Section 2 presents a scenario in which the use of ad-hoc transactional functionality is explained, as well as a short description of the principles behind PROSE and CheeTah. Section 4 discusses the architecture of the system and how CheeTah can be used as a transactional extension within PROSE. Section 5 presents a brief performance evaluation. Section 6 concludes the paper.

2 Motivation

2.1 Ad-Hoc Transactional Services

One of the most promising applications of the Internet is electronic commerce. This is also likely to be the case for spontaneous networks, where smaller and increasingly pervasive devices will engage in commercial transactions among themselves, with base stations, and with remote nodes reached through a combination of wireless and wired infrastructure.

Because nodes are mobile, there is a clear need for self-organization and adaptability. For example, a device might use electronic services in different locations where each location requires different behavior. One of the essential behavior adaptations is transactional functionality. An example of such an exchange is shown in Figure 1.a. The participating nodes could be in an open air trade fair where suppliers, manufacturers, retailers, and customer meet to see the latest products available. Customers may want to buy some of the products after seeing them and consulting an online electronic catalog made available at the fair. Upon receiving an order, the merchants can contact each other to order parts and locate supplies. With the appropriate devices, these orders can be registered and confirmed on site and later uploaded to the company's computers. More importantly, the transactional functionality guarantees overall correctness by ensuring that all parts for a product are ordered and the orders confirmed, that there are enough supplies of parts and products for the confirmed orders, etc. Since the system is ad-hoc, customers can consult and order from different vendors and vendors can contact different suppliers in a fully dynamic manner. New nodes (customers and merchants) can participate by simply entering the fair grounds. Nodes

can also leave without affecting other nodes (except that, of course, their services are no longer available).

The same scenario applies to nodes changing location. Assume there is a base station at each location (e.g., different fair grounds). Nodes that arrive to that location and want to execute transactions could receive the necessary code from the base station and then proceed as above. This code allows them to execute transactions with the base station and with any other node in that location. When they leave, this code is discarded. Upon arrival at a different location, a different version of the transactional code could be used in a similar manner. If there is no base station, the download could be done from some other node in that location.

To provide a secure environment for all services of a spontaneous network, it is essential to allow transactions of arbitrary complexity. The objective is that, once all nodes have the necessary software layer added to them, the interactions should be limited only by what transactional correctness allows.

2.2 Eliminating the Infrastructure: CheeTah and PROSE

One of the challenges of this scenario is to be able to guarantee transactional correctness without a centralized component. Based on *composite systems*, a solution to this problem has been recently developed (Cheetah [15]) and is now commercially available (Transactions Anywhere [3]). A composite system is a collection of autonomous service providers that interact transactionally following arbitrary configurations. Service providers reside at different locations and publish the services they are willing to provide. Components can offer increasingly complex services by combining and building upon the services provided by other components.

CheeTah is essentially a small TP-Monitor that resides in each node of the composite system. CheeTah aims to treat 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. CheeTah is very lightweight, less than 300 KBytes of code, and very flexible. However, the idea behind CheeTah was that applications running over the Internet will be developed using CheeTah and each node would have its own CheeTah server. In a spontaneous service community, we cannot assume that all nodes have CheeTah and, thus, we need a way to treat CheeTah as a runtime component that can be dynamically added to a given node. This implies having a method to systematically change the functionality of *all* service calls inside *all* nodes so that CheeTah is invoked before and after each service call. Such changes *cut across* the system, i.e., they cannot be easily located in a particular module or class.

Aspect oriented programming (AOP) [11] is one of the approaches proposed to address this problem. AOP allows the description of extensions to an existing application when these extensions cannot be easily expressed using traditional object-oriented techniques like inheritance. The description of such extensions is based on the concept of *aspects*, the part of a software system that affects the behavior of several components. An aspect defines a collection of points in the execution of a program. In AspectJ [20, 13], e.g., these points could be the invocations of some method(s) of a set of classes. In addition, each aspect contains the code to be inserted at (i.e., before or after) these execution points.

For providing transactional functionality, we need a solution that is capable of expressing cross-cutting adaptations at runtime. PROSE [17] is a dynamic AOP platform for Java, capable of detecting the events of interest and execute the additional actions (the extension) at runtime. Extensions are automatically applied every time a given event takes place (e.g., calling a method, modifying a field, etc.). PROSE is secure since it can use signatures to guarantee the integrity and authenticity of the code being added. It is also extremely flexible in that it allows to dynamically add and remove extensions to one or more Java-enabled nodes without affecting the underlying application.

In this paper, we show how to use PROSE to dynamically glue CheeTah to services exported by nodes joining a spontaneous network. This design allows us to consistently change the transactional behavior of a node as it forms new groups or enters different spaces.

3 Related Work

The explicit participation of the operating systems in the adaptation of underlying applications has been explored in the Odyssey system [14]. This form of adaptation is known as application-aware adaptation. This technique has been used, for instance, to hide the effects of mobility using replication and cache consistency techniques. Conceptually, this work is related to our approach since also advocates an active implication of the infrastructure in the adaptation of applications. The same need to shift a part of the adaptation logic away from the application has lead to approaches that propose new software architectures to support adaptive systems [8, 10, 16].

Agent systems like [6] define integrated infrastructures with support for distribution, coordination and dynamic behavior. Such systems are self-organized in the sense of providing the infrastructure for cooperative and distributed problem solving. Dynamic behavior features allow an agent to load a new program at run-time. The new program defines additional functionality that extends the agent's capability to react to messages. Our approach is related to dynamic agents in that it allows run-time instantiation of new components. However, it does not propose a new distributed computing model or a new service infrastructure. Its main focus is to allow *existing* service communities (we used Jini [2] as a prototype) to dynamically adapt to meet common goals. Thus, transactional correctness becomes possible due to CheeTah's ability to deal with arbitrary service configurations, while dynamic aspect-orientation [4, 9, 17, 18] allows services to be extended transparently.

4 Transactional Extension of Mobile Services

4.1 Basic Architecture

The problem of context-aware adaptation [19] usually addresses the problem of nodes physically present in a certain area or location. We assume that network connectivity (specifically, multicast-domains) is equivalent to co-location [7]. Thus, co-located nodes use the network infrastructure to publish services using a common service discovery and brokerage system [2, 12]. We further assume that all nodes are Java-based, and that they publish their services in the spontaneous network. Service invocations contain local operations (performed on a node-local database) and remote invocations to other nodes, discovered after joining a service community. In our architecture, the service discovery and brokerage functionality is provided by Jini [2] and service invocations are RMI based.

To illustrate self-organization in a service-community, consider a sub-interaction of the example scenario, in which three services are communicating (e.g., a merchant is ordering sub-parts from two different retailers). This situation is depicted in Figure 2.a. In a first phase (discovery), nodes publish the descriptions of the provided services while other nodes discover the services of interest (step 1). After having established the partners, an invocation that spans all three nodes A,B, and C is triggered by node A (step 2).

The self-organization part is represented by the adaptation of the nodes A, B, and C to the transactional requirements of the service community. This is achieved using PROSE extensions. These extensions are obtained from node D (Figure 2.b). Adaptation must be performed on *all* nodes of the service community expected to interact. In the fair-ground example, node D could be an available base-station, pre-configured to distribute fair-ground specific PROSE extensions. In a more dynamic setup, D may be one of the nodes, elected by other nodes to distribute the community-specific PROSE extensions. In the general case, one extension can be propagated from node to node as needed, while several extensions may be received from distinct nodes.

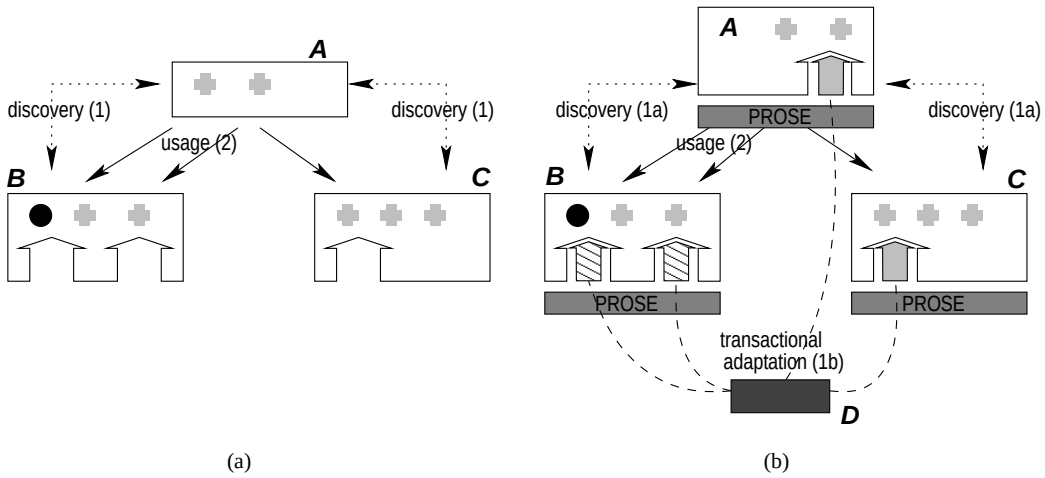


Fig. 2. (a) Service invocations in a pure spontaneous network (b) Service invocations in a spontaneous network with adapted nodes.

Extensions are distributed using the standard Java mechanism for downloading code. After discovery (step 1a), the insertion and removal of extensions is done by PROSE (step 1b). All participating nodes have PROSE running on them. With PROSE running, self organization and adaptation is achieved by exchanging extensions among the nodes. For transactional interaction we use CheeTah as an extension. The extension remain activated (and thus the service invocations are treated as transactions) until (i) the nodes leaves the service community (network disconnection), (ii) the transactional extension is replaced by a newer version, or (iii) the nodes autonomously decides to discard the extension.

When one node has received and activated several extensions, their effects may be overlapping or conflicting. In particular, when considering two different CheeTah extensions, we require that they intercept disjoint sets of remote service invocations. This avoids version incompatibilities.

4.2 Extensions

Extensions are compiled and signed Java classes. An extension describes where to do adaptations and what to do. Figure 3 contains an example of an extension. For reasons of space we cannot 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 [17].

```
1  class BeforeRemoteCall extends AspectExtension {
2    Crosscut createRoot = new FunctionalCrosscut() {
3     // if we are the root, create a transactional context
4     public void ANYMETHOD(ServiceB thisObj, REST params)
5        {
6           if ( lookup(TX_CONTEXT) == null)
7              {
8                 txCtx = ROOT_CTX;
9                 cheetahTM.bind(currentThread(),txCtx);
10             }
11       }
12     { setSpecializer(ClasseS.extending(Remote.class)). AND (MethodS.BEFORE) ); }
13   }
14 }
```

Fig. 3. A Java class containing the code for a runtime extension.

The extension is called BeforeRemoteCall. It contains one object (createRoot) which defines what to do (transform an invocation into a CheeTah transaction) and where to apply this action (before incoming remote calls of services of type B).

The method on line 4 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 CheeTah to be activated just before remote calls. This specialization is achieved on line 12 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 CheeTah to create a transaction context if the current invocation is not within the scope of a CheeTah transaction already (line 6). 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. 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.

4.3 Inserting and Withdrawing Extensions

In our system, each node runs on top of a JVM with PROSE activated. PROSE is exported itself as a Jini service, which makes possible the remote invocation of the addExtension method. To add the extension shown in Figure 3 to a an arbitrary number of services, the following code fragment has to be executed each time a node joins the spontaneous service community. Since our prototype is based on Jini, which supports service discovery, a concrete implementation of this action is straightforward.

```
1 proseProxy = getRecentlyDiscoveredNode(PROSE.class);
2 proseProxy.addExtension(new BeforeRemoteCall());
```

On line 1, the stub of the remote PROSE system is retrieved. On line 2, a new instance of the BeforeRemoteCall extension is created and then sent to the newly discovered node (e.g., node B) via its proxy. Immediately after the extension is activated on the remote node, PROSE will modify the execution for a potentially large number of method boundaries.

4.4 The CheeTah Runtime Extensions

With PROSE in place, ad-hoc transactional interaction is accomplished by using CheeTah as an extension. In an RMI call, we distinguish between the *caller* node (which performs the marshalling of parameters and unmarshalling of result and is responsible for the *outgoing remote call*) and the *called* node (which performs the unmarshalling of parameters, the method execution, and the marshalling of the results, and is responsible for the *incoming remote call*).

The transactional interaction is implemented by intercepting remote calls and determining whether they correspond to a root transaction (it is an remote call within a thread that is not associated with any transaction), or a sub-transaction (it is an remote call from a thread that already belongs to some transaction). With this information, CheeTah can take over and control the execution of the RMI calls as if they were nested transactions.

As an example, assume a node N_1 calls method m of a remote node N_2. By intercepting the call, the local CheeTah extension 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 RMI call to N_2, the CheeTah extension 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. Since the signature of the call cannot be modified (otherwise we may not be able to reconstruct the original call at the other end), the information CheeTah needs to propagate is sent as *hidden parameters* of the $N_2.m(arguments)$ RMI call.

At N_2, the invocation of m is intercepted. The local CheeTah extension at N_2 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 RMI 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) and its own node identifier (N_2). This is all the information a CheeTah local extension needs to detect the structure of a nested transaction [15].

When 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 [15]. As in the forward phase, the CheeTah extensions exchange information among themselves using a hidden *result* annotation. When a CheeTah extension sees that a call that just completed corresponds to a root transaction, this extension starts the 2-Phase-Commit protocol used to commit the results. The commitment protocol is performed among the extensions and does not involve intercepting calls.

All this behavior is implemented by adding two specific extensions to each node. The first extension is a RemoteCallContext extension. It provides the mechanism that allows a local CheeTah extension to communicate with a remote CheeTah extension without modifying the signatures of the calls being intercepted. On the caller side, this extension intercepts invocations to a node's communication layer (RMI) and sends the parent transactional data using mechanisms similar to the marshalling of arguments. On the called side, the information is extracted by intercepting the unmarshalling of arguments, thereby providing the data for the child transaction. Transactional information is also extracted when RMI calls return. This information allows to detect the termination of a sub-transaction.

The second extension is a TransactionalExtension. It relies on the existence of the extension RemoteCallContext. When this second extension is attached to a running node, it intercepts all remote invocations. If the extension, when notified of an incoming call, does not have any record of the node participating in a transaction, it immediately starts a root transaction. Otherwise, it associates the execution with a sub-transaction. Space constraints inhibit elaboration on the mechanisms used within CheeTah, specially those related to concurrency control and recovery; see [15] for details.

4.5 Transactional behavior in a service community

Once all nodes of the service community have been dynamically extended, computations involving several nodes become transactional CheeTah enforces a correctness criterion [1] for arbitrary, dynamic configurations of autonomous components. Thus, each participant can be involved in multiple transactions simultaneously, while preserving global correctness.

One important issue not discussed so far is how the system behaves if nodes leave the ad-hoc community while they are participating in a transaction. This problem is not particular to our system but a general problem of any distributed transaction. Thus, we rely on the functionality available in CheeTah for this purpose. If, for example, a non-root participant leaves the community, its caller will notice the failure of the sub-transaction corresponding to its last call. The caller application may try an alternative solution, e.g., to invoke another node providing the same service. When noticing a failed remote-call, the adaptivity layer implemented with PROSE will eliminate the subtransaction corresponding to the failed method from the overall transaction. The global transaction will not be aborted, due to the properties of nested transactions. While this heuristic

does not cover all possible failure cases, it provides a natural way to treat failures of participants. It may further happen that a participant leaves while it is in doubt about a transaction. While the time-window for such an event is small, the mini-tp-monitor implanted in the node will use an accepted policy of heuristic abort. Similarly, if the root node leaves the ad-hoc-community, the rest of the participants will remain in doubt until a time-out is reached. Then again, heuristic abort will be applied in each in-doubt participant. Such a heuristic requires manual intervention but it is a common approach in many database management systmes.

5 Performance Analysis of the Transactional Extensions

5.1 Description of the Experiments

The feasibility of using PROSE in real applications depends on the demands in terms of throughput and response time of the particular scenario. The scenarios we tested are similar to those described in Section 2. We have abstracted out some of the details of the application to test several configurations and to identify all potential bottlenecks.

In each experiment, a varying number of nodes implement services following the structure of a composite system. That is, a call to one of the services provided by one node triggers further calls to the services of other nodes which, in turn, may call yet other nodes, and so forth. Each service is implemented as a series of remote invocations (which become subtransactions once CheeTah is activated) and a number of local operations, including access to a database through a JDBC interface. All service invocations are implemented as RMI calls. In each experiment we vary the depth and width of the nested transaction involved. As it was done with CheeTah [15], we perform a worst case analysis in that all nodes in the system are equally loaded, i.e., a transaction goes through every node. This setup is not realistic since, in practice, transactions are likely to run in disjoint subsets of nodes. However, this artificially high load and conflict rate gives us a good idea of what can be achieved with the transactional extensions. If, under the worst case scenario assumption, the performance observed is acceptable, we can be sure that, under normal loads, the performance will also be acceptable. Note, nevertheless, that the type of ad-hoc network we have in mind is not likely to be used for high load, high volume transaction processing.

For the experiments, we used Pentium III 600 MHz Linux nodes. In a wireless LAN, the limited bandwidth is the determining factor in terms of throughput. Thus, to avoid the network bottleneck, we assume sufficient bandwidth by performing the experiments on a 100 Mbps Ethernet LAN. For data storage, we used an Oracle 8.0.3 RDBMS. The number of concurrent clients is determined by the configuration used in the experiment. Each configuration was characterized by the height of the root invocation hierarchy (how many levels until the leaves are reached, the leaves being the access to the local database) and its width (the number of sub-transactions per parent transaction). Figure 4.a shows an example of a configuration with depth 3 and width 1. Such a configuration is denoted 3x1. Figure 4.b is a 2x3 configuration, with the root invoking three other services (width is 3) and a total depth of 2. The configurations considered in the experiments include: 1x1, 1x2, 1x3, 1x4, 2x1, 3x1, 4x1, 2x3, 2x4.

Fig. 4. (a) 3x1 configuration with 2 levels and width 1,(b) 2x3 configuration with 2 levels and width 3.

Following the worst case scenario analysis idea, in all experiments we used a varying number of clients that connect to the very same node. This node acts as the root for all transactions, which

adds further overhead. In this way, we can measure how far a given service can be loaded. Note that the node acting as the root has a significant load since, for instance, it has to initiate the commit protocol and create the context for all root transactions. From the transactional point of view, the extension added used the compensation mechanism provided by CheeTah.

Table 1 shows the results of the experiments for the different configurations. The two rows summarize the response time and throughput for CheeTah running as a PROSE extension.

Configuration	1x1	2x1	3x1	4x1	2x2	2x3	2x4
Average Response Time	0.1	0.17	0.22	0.24	0.22	0.26	0.33
Transactions/minute	615	344	271	251	275	233	181

Table 1. Response time (in seconds) and transactions per minute for CheeTah running as an extension.

In terms of throughput, the depth of the transaction plays a significant role in the overall performance (as was observed by Pardon and Alonso [15]). As the results for configurations 1x1, 1x2, 1x3, and 1x4 show, the deeper the transactions, the smaller the throughput and the larger the response time. Since a deeper transaction involves more RMI calls, these results are to be expected. In all cases, the overhead incurred by the dynamic interception mechanism, implemented with PROSE, ranged between 1.3% and 2% of the response time.

6 Discussion and Conclusions

Although the results presented are preliminary, they prove to a great extent the feasibility of the idea. One important issue not discussed so far is how the system behaves if nodes leave the ad-hoc community while they are participating in a transaction. This problem is not particular to our system but a general problem of any distributed transaction. Thus, we rely on the functionality available in CheeTah for this purpose. If, for example, a participant leaves while it is in doubt about a transaction, the mini-tp-monitor implanted in the node will use an accepted policy of heuristic abort. Such a heuristic requires manual intervention but it is a common approach in many database management systems.

Obviously, we do not expect to get the same high performance as that reached in the experiments. However, the experiments do show that dynamic adaptation is possible even in the case of transactional functionality. From a practical point of view, the initial prototype has allowed us to gain some experience and will form the basis to explore more elaborate application scenarios. Despite its limitations, it provides an interesting starting point to explore a part of the design space of future information systems. For instance, the scenarios we have in mind will be limited by the interactive nature of the applications and the low bandwidth of wireless settings. These two aspects will be much stronger constraints that PROSE and CheeTah so we are certain that the overhead introduced by our approach is acceptable in practice. Hence, since transactional interactions on ad-hoc networks are a key requirement for many applications, we believe the prototype described provides a realistic solution for this service. As the next step, we are currently porting the entire system to hand-held devices and will soon start the testing phase.

References

1. G. Alonso, A. Fessler, G. Pardon, and H.-J. Schek. Correctness in general configurations of transactional components. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS'99)*, Philadelphia, PA, May 31 - June 2 1999.
2. K. Arnold, A. Wollrath, B. O'Sullivan, R. Scheifler, and J. Waldo. *The Jini Specification.* Addison-Wesley, Reading, MA, USA, 1999.
3. Atomikos. Peer-to-peer distributed transaction processing and distributed databases (iCatch). www.atomikos.com, 2002.
4. J. Baker and W. Hsieh. Runtime Aspect Weaving Through Metaprogramming. In *1st International Conference on Aspect-Oriented Software Development, Enschede, The Netherlands*, April 2002.

5. K. Boucher and F. Katz. *Essential guide to object monitors.* John Wiley & Sons, 1999.
6. Qiming Chen, Parvathi Chundi, Umeshwar Dayal, and Meichun Hsu. Dynamic-agents for dynamic service provisioning. In *Proceedings of the 3rd IFCIS International Conference on Cooperative Information Systems, New York City, New York, USA, August 20-22, 1998*, pages 95–104. IEEE Computer Society, 1998.
7. P. Couderc and A.-M. Kermarrec. Enabling context-awareness from network-level location tracking. *Lecture Notes in Computer Science*, 1707, 1999.
8. C. Efstratiou, K. Cheverst, N. Davies, and A. Friday. An Architecture for the Effective Support of Adaptive Context-Aware Applications. *Lecture Notes in Computer Science*, 1987, 2001.
9. N.D. Hoa. Dynamic Aspects in SOFA/DCUP. Technical Report 99/07, Charles University, Prague, June 1999.
10. E. Kiciman and A. Fox. Separation of Concerns in Networked Service Composition. Position Paper Workshop on Advanced Separation of Concerns in Software Engineering at ICSE 2001, Toronto, Canada, May 2001.
11. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, NY, June 1997.
12. 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.
13. Cristina Videira Lopes and Gregor Kiczales. Recent Developments in AspectJ. In Serge Demeyer and Jan Bosch, editors, *Object-Oriented Technology: ECOOP'98 Workshop Reader*, volume 1543 of *Lecture Notes in Computer Science*, pages 398–401. Springer, 1998.
14. B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile Application-Aware Adaptation for Mobility. In *Sixteen ACM Symposium on Operating Systems Principles*, pages 276–287, Saint Malo, France, 1997.
15. G. Pardon and G. Alonso. CheeTah: a Lightweight Transaction Server for Plug-and-Play Internet Data Management. In *Proceedings of VLDB 2000*, Cayro, Egypt, September 2000.
16. S. R. Ponnekanti, B. Lee, A. Fox, P. Hanrahan, and T. Winograd. ICrafter: A Service Framework for Ubiquitous Computing Environments. *Lecture Notes in Computer Science*, 2201, 2001.
17. A. Popovici, T. Gross, and G. Alonso. Dynamic Weaving for Aspect Oriented Programming. In *1st International Conference on Aspect-Oriented Software Development, Enschede, The Netherlands*, April 2002.
18. L. Duchien R. Pawlak, L. Seinturier and G. Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, pages 1–24, Kyoto, Japan, September 2001. Springer Verlag.
19. B. Schilit, N. Adams, and R. Want. Context-Aware Computing Applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, US, 1994.
20. Xerox Corporation. The AspectJ Programming Guide. Online Documentation, 2001. http://www.aspectj.org/.