Consistent Process Execution in Peer-to-Peer Information Systems

Klaus Haller and Heiko Schuldt

Database Research Group, Institute of Information Systems
Swiss Federal Institute of Technology (ETH Zürich)
CH-8092 Zürich, Switzerland {haller,schuldt}@inf.ethz.ch

Abstract. The proliferation of Internet technology resulted in a high connectivity between individuals and companies all over the world. This technology facilitates interactions within and between enterprises, organizations, etc. and allows for data and information exchange. Automating business interactions on this platform requires the execution of processes. This process execution has to be reliable, i.e., guarantees for correct concurrent and fault tolerant execution are vital. A strategy enforcing these properties must take into consideration that large-scale networks like the Internet are not always reliable. We deal with this by encapsulating applications within mobile agents. Essentially, this allows users to be temporary disconnected from the network while their application is executing. To stress the aspect of guarantees, we use the term *transactional agents*. They invoke services provided by resources, which are responsible for logging and conflict detection. In contrast, it is the transactional agents' task to ensure globally correct concurrent interactions by means of communication. The used communication pattern is a sample implementation of our newly developed protocol. It is, to our best knowledge, the first distributed protocol that addresses the global problem of concurrency control and recovery in a truly distributed way and that, at the same time, jointly solves both problems in a single framework. Because (i) processes are long running transactions requiring optimistic techniques and (ii) large networks require decentralized approaches, this protocol meets the demands of process-based applications in large scale networks.

1 Introduction

The proliferation of Internet technology in the recent years resulted in a high connectivity between individuals and companies all over the world. Together, they form a unique world-wide platform for business interactions, or – more general – for data and information exchange. Hence, this technology facilitates interactions within and between enterprises, organizations, etc. This offers new opportunities for automating business interactions, because a wide variety of services – like booking flights and hotels or registering for conferences – are today supported by computers and can be invoked over the network, e.g., the Internet. Application development in this context requires to combine different invocations of such services into a coherent whole. This is done by using *processes*. Each activity of a process corresponds to a service invocation or, in case of intra-process parallelism, to the invocation of a set of semantically equivalent services.

J. Eder and M. Missikoff (Eds.): CAiSE 2003, LNCS 2681, pp. 289–307, 2003.
© Springer-Verlag Berlin Heidelberg 2003

Vital to such process-based applications in large-scale dynamic networks like the Internet is to have sophisticated system support guaranteeing that executions are reliable. Essentially, process executions have to be shielded from unreliable network connections. However, several problems resulting from this kind of environment have to be solved in order to provide a holistic solution for reliable process execution.

First, the dynamics of the network has to be taken into account. This means that the location of services to be invoked as process activities is not known at built-time when processes are specified (*dynamic service providers*). Hence, it is not possible to hard-code this information. Consequently, the services corresponding to process activities have to be specified by using predicates. In a travel planning process, for instance, an activity should rather be specified like 'Go to all European airlines and search for flights from Zürich to Klagenfurt' instead of coding this in the traditional way 'Go to `www.swissair.com` and `www.aua.com` and invoke appropriate services (whose names have also to be known at built-time) there'. Obviously the second approach has disadvantages in a dynamic world where certain airlines stop operation or, more importantly, new low-price airlines start operation.

Second, the characteristics of the underlying network have to be considered. Essentially, when the *network is unreliable*, e.g., when network partitioning takes place or when bottlenecks occur, this has severe effects on the execution of processes. If, for instance, the travel planning process is executed on a mobile device like a PDA, this device has to be connected to the network as long as the process is executing. Otherwise, process execution would stop since remote services could no longer be invoked. The same problem arises in case of network partitioning if a process wants to invoke a service that belongs to a different partition. To cope with this problem, both process descriptions and their execution states have to be brought to the hosts on which the services reside. To do this, we use the concept of mobile agents [Whi97]. Processes are encapsulated in a single mobile agent which migrates through the network for the purpose of process execution. In the case of intra-process parallelism, the mobile agent clones such that these clones are able to invoke different services at the same time even at different hosts. In terms of the travel planning application started on a PDA, this paradigm gives the user the freedom to disconnect from the network after her application is launched. For all these reasons, mobile agents seamlessly provide support to problems related to unreliable network connections.

Third, since service providers are autonomous, they can unilaterally decide to revoke their services. This might have severe consequences on process executions when among these services a particular service has been unique in the network (i.e., when there is no semantically equivalent service left). Essentially, all processes in which this service is to be used cannot continue their execution. However, in order to reliably execute processes, each process has to terminate in a well-defined state, even in the case of failures of the underlying network, in the case services disappear, or when run-time failures of process instances occur. Yet, all these failures have to be handled in a flexible way. Essentially, failure handling for processes must not be realized with an "all-or-nothing" semantics meaning that all effects of a process are undone after a failure occurs. Rather, following the idea of transactional processes [SABS02] which support semantic atomicity [GM83], appropriate alternative execution strategies should be considered. These alternative strategies have to be specified at built-time. Most importantly, they

allow for flexible failure handling by forward recovery [Tay86]. In the travel planning example, such a contingency strategy could be to search for a train connection when no appropriate flight is offered or when all seats are already booked.

Finally, different processes may concurrently access shared data, wrapped by services. In this case, appropriate mechanisms are required to protect processes to interfere with each other in an undesired way. Obviously, this *isolation* property requires that flow of information between independent processes via shared services is controlled such that a consistent view of the overall system is guaranteed. Therefore, the system is responsible for controlling the concurrent access to shared services and, as a consequence, to correctly synchronize parallel processes. The latter two problems, i.e., (semantic) atomicity and isolation, are well understood in traditional database systems. However, the first two aspects mentioned above – unreliable network connections and dynamically evolving networks– require to re-think the solutions that have been proposed so far. Conventionally, transaction management supporting atomic and isolated executions relies on a centralized coordinator, i.e., a transaction manager.

However, such a centralized approach is not suitable in large-scale, unreliable and dynamic networks. First, having a dedicated place for controlling all applications within the overall network might soon become a bottleneck when the number of processes increases. Second and more importantly, processes are not able to execute correctly in case the coordinator cannot be contacted due to some network failure. For all these reasons, a solution for the problems related to failure handling and synchronization has to be combined with mobile agents we use as operational environment for process execution. Support for flexible failure handling requires that alternative execution strategies are made available to these agents as parts of the process specification such that these alternatives can be effected in case of failures. While this can be seamlessly solved locally within each agent, support for the global problem of synchronizing concurrent process executions requires a more radical shift of paradigm. The synchronization of concurrent applications is shifted to the individual agents. Hence, the global problem is solved by distributing meta information needed for synchronization purposes to the individual agents. This meta information is then kept consistent by means of communication. In particular, it allows to locally decide whether multi-agent executions are correct.

The combination of all these aspects leads to a framework supporting *information processing in peer-to-peer environments*. We ensure that processes – which not only search for and retrieve data, but also manipulate data – are executed in peer-to-peer networks with transactional guarantees. The main contribution of this paper is a decentralized approach to process synchronization. This approach is embedded into an operational environment in which mobile agents flexibly execute processes in dynamic peer-to-peer networks. Within the AMOR project (Agents, MObility, tRansactions) of ETH Zürich, we have built a prototype system that accounts for all different aspects.

The paper is organized as follows: We start with an overview of the AMOR system (Section 2), followed by a discussion of the decentralized transaction processing architecture in Section 3. Section 4 discusses the main ideas underlying the AMOR protocol. The latter will be presented in detail in Section 5. We discuss the AMOR-prototype in Section 6. Section 7 provides a survey of related work. Section 8 concludes.

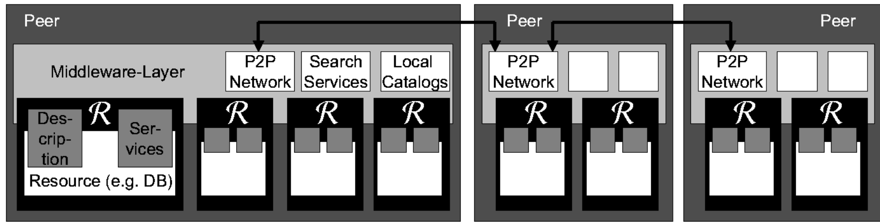

Fig. 1. The AMOR System Model

2 The AMOR System

The bottom layer of the AMOR system encompasses a set of distributed and autonomous *resources*, e.g., database systems or flat files (Figure 1). Such resources are wrapped by so-called *resource agents* ($\mathcal{R}$). Each resource provides a certain set of services to the outside world. This allows to hide the details of the type of resources and the implementation of services. Thus, by using a service-oriented interface, data access and manipulation takes place at a semantically higher level of abstraction compared to traditional read/write operations in databases. However, in order to make these services available, each resource agent has to provide metadata describing its type and the semantics of its resource, e.g., whether it manages flights or databases. More formally, we denote by $\mathcal{S}^*$ the universe of all services offered and, accordingly, the universe of all resource agents in our system by $\mathcal{R}^*$. We assume the resources wrapped by all $\mathcal{R}_i, \mathcal{R}_k \in \mathcal{R}^*$ to be pairwise disjoint. With this, we demand that the services of a resource agent only operate on local resources and are not redirected to remote resources.

All resources and therefore also the resource agents reside on *peers*[1]. Hosts may accommodate more than one resource agent. Also, the number of resource agents may differ between peers. Peers communicate with each other via the AMOR middleware layer (Figure 1). This layer forms a peer-to-peer (P2P) network out of the single peers. Each of them can unilaterally and spontaneously decide to join or leave the network such that the network configuration continuously evolves.

On top of this peer-to-peer network, mobile *transactional agents* execute processes. Each transactional agent corresponds to a process instance. Processes consist of a set of partially ordered activities. Activities, in turn, correspond to service invocations wrapped by some $\mathcal{R}_i$. When specifying a process, a programmer has to describe the type of services that are to be invoked as process activities. In addition, certain activities allow for several semantically equivalent services to be invoked in parallel. For example, finding the cheapest flight to some destination may take place by contacting several resources concurrently. Hence, it is the task of the programmer to set the level of concurrency for individual activities. All this information is the input for an agent generator which assembles a mobile transactional agent (Figure 2). This agent executes the process by migrating from peer to peer within the network and by locally invoking services. More formally, a transactional agent $\mathcal{T}_i$ is a pair $(S_i, <_i)$ where $S_i \subseteq \mathcal{S}^*$ is a set of services

[1] In the mobile agents terminology, they are also called *places*.

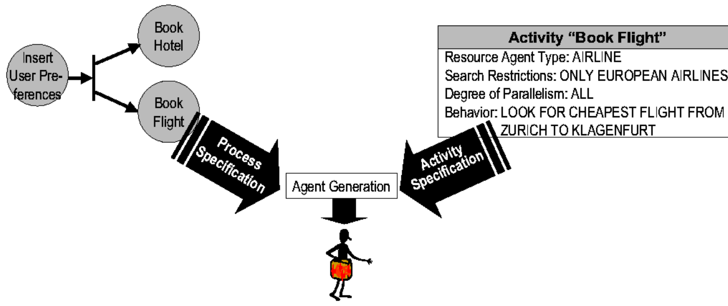

Fig. 2. Agent Generation Process

invoked by $\mathcal{T}_i$ and $<_i \subseteq S_i \times S_i$ is the (partial) invocation order of the services of S_i with c_i (commit) or a_i (abort) as terminal elements.

The invocation of services requires that services which meet the specified semantics can be found dynamically. Hence, the run-time environment has to support the search for particular services by providing service repositories where services are described for example by WSDL. This search functionality is incorporated into the peer-to-peer middleware layer. This layer evaluates the specification of an activity (Figure 2) to be executed next and returns the location of appropiate $\mathcal{R}$s. Then – if parallelism is desired – the agent clones and each clone migrates to a dedicated $\mathcal{R}$. After all prallel service invocations have succeeded, the next activity, according to the process specification, is executed. Again, the agent searches for potential peers (possibly clones) and migrates to the peer where the service invocaction takes place. In case of a failure, the system should be brought back to a consistent state. Additionally – to enforce consistency – also the flow of information between processes has to be controlled in order to enforce consistency.

3 The AMOR Approach for Decentralized Transaction Processing

In this section, we present the basic concepts of the AMOR decentralized approach to transaction management. This includes a comparison with the conventional transaction management architecture. Thereby, we focus on how to enforce the isolation property in AMOR. In AMOR, each $\mathcal{T}_i$ represents an individual, independent, and distributed transaction. In such environments, transaction management is typically provided by a dedicated coordinator, e.g., a TP Monitor [BN97]. The coordinator's task is to orchestrate the execution of distributed transactions, in particular, to enforce not only their atomic commitment but also their isolated execution by using a 2PC/2PL-protocol [GR93]. This requires all transactions of the system to register with the coordinator such that the latter one is equipped with global knowledge.

Such a centralized approach is highly appropriate for small, well-delimited environments with a fixed number of peers, but it cannot be applied to large-scale networks with a large number of distributed, heterogeneous, and autonomous peers (e.g., the intranet of an international company, not to speak about globally distributed networks like the

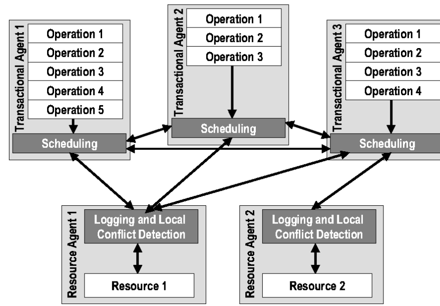

Fig. 3. Decentralized AMOR Approach to Transaction Processing

semantic web). In addition, also administrative aspects (i.e., peers might dynamically join or leave the system) prohibit centralized approaches to be used efficiently.

Hence, we must strive for a *distributed implementation* of the functionality of a coordinator. Following the unified theory of concurrency control and recovery [VHBS98], this coordinator jointly enforces both atomicity and isolation. In terms of recovery, each failure of some $\mathcal{T}_i$ has to be handled properly, either by an alternatvie execution or by undoing the effects of all services it has invoked so far (semantic atomicity). If $\mathcal{T}_i$ wants to undo activities, it has to know where they have been executed. Consequently – because there is no centralized component to which all requests are submitted – $\mathcal{T}_i$ itself has to log all its service invocations. Using this log information, a $\mathcal{T}_i$ can undo the effects of its regular services in reverse order (or, if specified in the process model, sets of entities can be undone by additional compensation activities). Hence, the abort of a $\mathcal{T}_i$ is followed by service invocations compensating the previous, regular invocations and finally by the commit of c_i. By this, we can enforce the atomicity property. In addition, the concurrent invocation of services has to be controlled such that each $\mathcal{T}_i$ is correctly shielded from other transactional agents. Shielding means in this context that there is no cyclic flow of information between different transactional agents.

Such a flow of information emerges if different $\mathcal{T}_i$s invoke non-commuting services. Two services do not commute (are in conflict) if their order matters, i.e., if the return values of the two service invocations or those of any succeeding activities change when the conflicting activities were ordered differently. More formally, the invocations of two services $s_{k,j}$ and $s_{k,m}$ provided by resource agent $\mathcal{R}_k$ commute if the return values of all services in the sequence $S' < s_{k,j} < s_{k,m} < S''$ are the same than those in the sequence $S' < s_{k,m} < s_{k,j} < S''$ where S' and S'' are arbitrary sequences of service invocations from $\mathcal{S}^*$ [Aea94]. This conflict behavior belongs to the knowledge of each $\mathcal{R}$. So each $\mathcal{R}$ can reason about its local conflicts. The union of all local conflicts of $\mathcal{R}$ comprises – due to the independence of the resources – all the conflicts of the system.

Conflict information is needed for reasoning about global correctness. This reasoning is based on the notion of schedule. Such a schedule $\mathbf{S}$ is a tuple $\mathbf{S} = (\mathcal{T}_S, <_S)$ reflecting the concurrent execution of a set of transactions $\mathcal{T}_s$ where $<_S$ is the order in which the services of the $\mathcal{T}_i \in \mathcal{T}_S$ are invoked (with $<_i \subseteq <_S$ for all $\mathcal{T}_i$ of $\mathcal{T}_S$).

Using a serialization graph for reasoning about the correctness of a schedule requires global information to be available at a central coordinator — the transaction manager.

However, due to the absence of such a transaction manager in AMOR, conflict information on local service invocations has to be communicated from a resource agent to the corresponding $\mathcal{T}_i$. Then, scheduling can be enforced by distributing this conflict information between transactional agents, thereby replicating meta information needed for synchronization purposes. In Figure 3, this distributed AMOR approach to transaction management is illustrated.

Yet, an important aspect of the AMOR protocol is to bridge the gap between the available, local view of transactional agents and the global knowledge needed to enforce the correctness criteria of the unified theory of concurrency control and recovery. Given the absence of a global coordinator, the challenge is to nevertheless enforce global correctness, although transactional agents are acting autonomously and in parallel while not necessarily having up-to-date information on the services invoked by other transactional agents.

4 Concepts and Data Structures for Decentralized Transaction Processing in AMOR

4.1 Ensuring Correctness at Commit-Time: Requirements and Limitations

The applications implemented by mobile transactional agents are considered to be highly distributed and long-running, thereby far exceeding the complexity and duration of conventional funds transfer transactions. Hence, traditional locking-based protocols, e.g., strict two phase locking, cannot be applied. The reason is that these protocols would – especially when combined with two phase commit protocols – unnecessarily block the access to the underlying resources. In contrast, due to the independence of resources, conflicts may only occur when two transactional agents invoke services on the same resource. Business processes are often characterized as applications for which conflicts are rare. AMOR is supporting this characterization, although it is not a vital requirement for the AMOR protocol.

For all these reasons, we follow an optimistic approach in AMOR. This means that each $\mathcal{T}_i$ executes a service invocation of $\mathcal{T}_i$ on the spot without determining whether is allowed or not. However, prior to the commit of $\mathcal{T}_i$, a validation is required which checks whether it has executed correctly (based on the correctness criteria of the unified theory) and is therefore allowed to commit. This is closely related to well-established optimistic concurrency control protocols like backward-oriented concurrency control, BOCC [KR81]. Following this optimistic protocol, a $\mathcal{T}_i$ is allowed to commit if it does not depend on some active (uncommitted) $\mathcal{T}_j$. In here, dependency means that $\mathcal{T}_i$ and $\mathcal{T}_j$ are in conflict at some resource $\mathcal{R}_k$ by services $s_{k,i}$ invoked by $\mathcal{T}_i$ and $s_{k,j}$ invoked by $\mathcal{T}_j$ with $s_{k,j} < s_{k,i}$. In order to guarantee that no such dependencies exist, each $\mathcal{T}_i$ has to query 'its' resource agents at commit time (i.e, the resource agents where it has invoked services). Then, the latter determine – based on their local log – all local conflicts with active transactional agents in which $\mathcal{T}_i$ is involved. Hence, a transactional agent is allowed to commit correctly i.) when it has queried all resource agents where it has invoked services, ii.) when all have responded to this query, and iii.) when there is no dependency to an uncommitted transactional agent.

While this approach enforces correct concurrency control, it features an important drawback. Essentially, cyclic conflicts and therefore non-serializable executions are not detected until commit time, even when these cyclic conflicts have been imposed much earlier in the work of some $\mathcal{T}_i$. Yet, detecting such cyclic conflicts at an earlier stage, i.e., when they actually appear, would allow for much less redundancy.

4.2 Region Concept

In order to overcome the drawback of enforcing correctness at commit time, we have to detect cycles as early as possible. Thus, we equip each $\mathcal{T}_i$ with metadata reflecting a multi-agent execution. Such metadata – in a first approach – could be a copy of the global serialization graph. This, on the one side, allows each transactional agent to detect cyclic conflicts early. But on the other side, maintaining a copy of the complete serialization graph with each $\mathcal{T}_i$ is not practical since it would impose considerable communication overhead.

However, an important observation is that in the type of system we are addressing, more or less closed –albeit not static– communities exist. These communities are sets of closely related resources accessed by the same transactional agents, i.e., agents which aim at addressing the same or similar tasks. Hence, conflicts are only possible between agents invoking services within the same community. We denote the set of transactional agents executing in such a community as *region*. In terms of the data structures maintained for concurrency control purposes, a region corresponds to the nodes of a connected subgraph of the global serialization graph. Consequently, only this connected subgraph (termed *region serialization graph*) has to be replicated among all $\mathcal{T}_i$ of the same region. Obviously, if all region serialization graphs are free of cycles, so is the global serialization graph. Thus, reasoning about system-wide correctness can be safely shifted to the universe of region serialization graphs. By making use of the partitioning of the system in disjoint regions, AMOR uses replicas of region serialization graphs maintained by each transactional agent to enforce correct multi-agent executions. This requires that i.) conflicts are communicated from resource agents to the corresponding transactional agents and ii.) the replica of region serialization graphs are kept consistent among all $\mathcal{T}_i$ of the same region.

However, regions are not static. Rather, the composition of regions is affected by the services invoked by the transactional agents of a region. For example if a service invocation of $\mathcal{T}_i$ imposes a conflict with a service invocation of a $\mathcal{T}_j$ of another region – as a consequence of this conflict – the two originally independent regions have to be merged.[2] Hence, as an additional requirement for consistent metadata management in AMOR, the two region serialization graphs have to be consolidated and finally distributed among all transactional agents of the new region. Similarly, by correctly committing a transactional agent or in case of a rollback, regions may split and so does the corresponding region serialization graph.

[2] In worst case, when all $\mathcal{T}_i$ conflict, there is only one region and the complete serialization graph has to be replicated.

4.3 Region Serialization Graphs

The task of guaranteeing consistency of replicated region serialization graphs is shifted to the transactional agents of the corresponding region. The information needed for this purpose is provided by the resource agents (such that each $\mathcal{T}_i$ is able to update its local copy of the region serialization graph after a service invocation) and is exchanged between transactional agents by means of asynchronous messages so as to keep the local replica consistent. Every time the local region serialization graph of some $\mathcal{T}_i$ changes due to a service invocation, the complete graph is propagated to all other $\mathcal{T}_j$ of the same region. Essentially, communication between transactional agents has to consider delays in the delivery of messages leading to message overtaking. Therefore, certain extensions to the data structure of region serialization graphs are required.

Consistently with the traditional notion of serialization graph, the nodes of the region serialization graph correspond to transactional agents and the directed edges to conflicts between them. However, the edges are marked with the pairs of service invocations that have caused the edge (note that several such pairs might exist for the same edge). In addition, *each edge* is extended by a version number. However, changes of the serialization graph are communicated by messages. So message overtaking has to be taken into account for instance when messages are delivered along different paths. Then, it is essential to know which information is more up to date, when region serialization graphs are merged after a new graph has been received from another transactional agent.

5 A Protocol for Decentralized Optimistic Transaction Processing

The aim of the AMOR protocol is to enforce correct concurrent and fault tolerant agent execution. This requires communication between transactional agents in order to exchange meta-data regarding the order of conflicting service invocations. The latter is derived from the resource agents. In order to let the transactional agents know about these conflicts, the resource agents have to report local conflicts to them after services have been invoked.

5.1 Service Invocation

Processes are executed by transactional agents $\mathcal{T}_i$. Therefore, they contact resource agents $\mathcal{R}_k$ in order to invoke services $s_{k,i}$ on them. Such services are – due to the optimistic approach followed in AMOR – executed immediately. At the same time, not only each transactional agent logs where it has executed a service, but also the resource agents make a corresponding log entry. Using the local log and the local commutativity relation, each resource agent determines if and which new conflicts have emerged.

In case of new conflicts, $\mathcal{R}_k$ returns information about these conflicts to $\mathcal{T}_i$. This notification contains information on all $\mathcal{T}_m$ this service invocation is in conflict with and the services of these $\mathcal{T}_m$. Then, $\mathcal{T}_i$ – as the receiver of this information – first updates its local region serialization graph: If the graph does not contain a newly emerged edge up to now, meaning that the represented dependency is not known, it is inserted in the graph and the version number (see Section 4.3) is set to one. Otherwise, if the edge

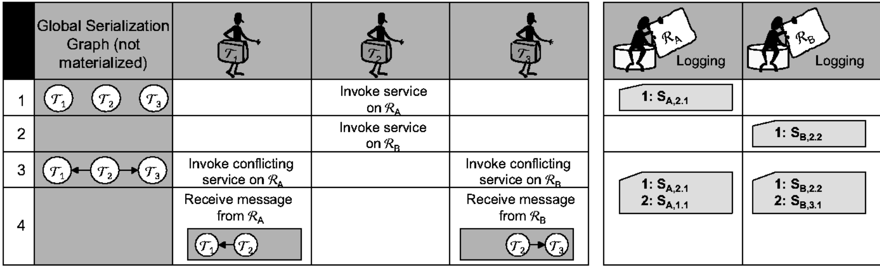

Fig. 4. Example: Service invocation

is marked as removed, it means that such a conflict has existed at least once before. However, it disappeared later because of a partial rollback. Consequently, now, the edge is marked as valid and the version counter increased by one. After $\mathcal{T}_i$ has updated its local graph, it must share this information with the other transactional agents. Therefore, the serialization graph is distributed (see below), before the next service is invoked.

We want illustrate the service invocation procedure with a small sample scenario. In this scenario, we assume two resource agents, $\mathcal{R}_A$ and $\mathcal{R}_B$. On top of them, three transactional agents, $\mathcal{T}_1, \mathcal{T}_2$, and $\mathcal{T}_3$ operate. This execution (and the side-effects on the system state) are illustrated in Figure 4. The left column of this figure depicts the global system information, i.e., the global serialization graph. This information is, however, not permanently materialized by any agent of the system. The other columns contain knowledge materialized in the different agents and the activities performed by the transactional agents.

In the first two steps, $\mathcal{T}_2$ invokes a service on $\mathcal{R}_A$ and one on $\mathcal{R}_B$, respectively. In the third step, the two other transactional agents $\mathcal{T}_1$ and $\mathcal{T}_3$ concurrently invoke services on $\mathcal{R}_A$ and $\mathcal{R}_B$, respectively, which both conflict which the ones of $\mathcal{T}_2$. This is detected by $\mathcal{R}_A$ and $\mathcal{R}_B$. Following the AMOR coordination protocol, they notify $\mathcal{T}_1$ and $\mathcal{T}_3$ about these local conflicts such that $\mathcal{T}_1$ and $\mathcal{T}_3$ can maintain their local region serialization graphs (step 3). Nevertheless, none of the local serialization graphs contains the full knowledge of the (not materialized) global serialization graph: While $\mathcal{T}_2$ is not aware of any conflict, both other transactional agents only know about the conflicts they are involved in. Consequently, the transactional agents have to exchange their information.

5.2 Messaging: Distribution of Metadata

Enforcing consistency between local serialization graphs and the global graph requires that the transactional agents exchange messages. For this, an agent $\mathcal{T}_i$ which has caused a new edge by some service invocation (or any other change of the graph) is responsible for distributing this information within its region, since $\mathcal{T}_i$ is the only one being immediately informed about this by the underlying resource agent.

When a $\mathcal{T}_p$ receives a message from $\mathcal{T}_i$ containing the extended region serialization graph of $\mathcal{T}_i$, it updates its local version of the graph. Thus, $\mathcal{T}_p$ not just replaces its copy with the newly arrived one, but it has to merge the two versions. This is due to the fact that different transactional agents may concurrently invoke services. Consequently,

they concurrently update their local copy of the serialization graph. A second reason is that it has to be ensured that newly arrived, yet older data does not overwrite formerly received, newer data during the process of merging. In case an edge appears in both graphs, the one with the higher version number is chosen. Afterwards – to prevent the graph from uncontrolled enlargement – it is checked for nodes that do not belong the region anymore. Such edges can be safely removed.

After merging, $\mathcal{T}_p$ propagates the merged graph to all transactional agents which have been in its local copy but not in the one received from $\mathcal{T}_i$. So they also receive the new information. Hence, also $\mathcal{T}_i$ may be a receiver in case $\mathcal{T}_p$ has knowledge not known to $\mathcal{T}_i$ (e.g., in case of the consolidation of two different regions). The recipients of the message originating from $\mathcal{T}_p$, in turn, merge the new graph with their local replica and again forward it, if needed.[3]

This forwarding and graph merging is the subject of the continuation of our example in Figure 5. After the update of the local serialization graphs, the propagation process is started. Since both $\mathcal{T}_1$ and $\mathcal{T}_3$ have caused changes to their region serialization graph, they have to inform all other transactional agents of the same region (step 5). For this, they depend on their local knowledge about the region. So $\mathcal{T}_1$ and $\mathcal{T}_3$ both only send a message to $\mathcal{T}_2$ encompassing their updated region serialization graph, but not to each other, because both do not know that the other agent also belongs to the same region. In the following step 6, we assume that the message from $\mathcal{T}_1$ arrives at $\mathcal{T}_2$. So $\mathcal{T}_2$ is able to update its graph with the new information. Because $\mathcal{T}_2$ does not have any new information, there is no need to send any message back to $\mathcal{T}_1$. Note that $\mathcal{T}_2$ cannot propagate its graph to $\mathcal{T}_3$, because it does not know about it up to now.

This changes in step 7, when $\mathcal{T}_2$ receives the message of $\mathcal{T}_3$. So $\mathcal{T}_2$ merges the newly received graph from $\mathcal{T}_3$ with its local one. Obviously, simply overwriting the local copy of $\mathcal{T}_2$ is not the correct solution, because information about the conflict between $\mathcal{T}_2$ and $\mathcal{T}_1$ would be lost. But if $\mathcal{T}_2$ merges both graphs, this results in a region serialization graph which now encompasses information on all conflicts of the region.

Afterwards, $\mathcal{T}_2$ has to figure out whether or not it has to propagate its region graph. Because the message sent by $\mathcal{T}_3$ did not contain any edge between $\mathcal{T}_1$ and $\mathcal{T}_2$, $\mathcal{T}_3$ obviously has less information than $\mathcal{T}_2$ and has to be a receiver. The same is true for $\mathcal{T}_1$. So $\mathcal{T}_2$ assumes that $\mathcal{T}_1$ has the same knowledge. Consequently, $\mathcal{T}_2$ reasons that $\mathcal{T}_1$ does not know about the edge between $\mathcal{T}_2$ and $\mathcal{T}_3$. This results in the situation in step 8, in which $\mathcal{T}_2$ forwards its merged graph to $\mathcal{T}_1$ and $\mathcal{T}_3$. Again, these messages contain the complete region serialization graph of the sender ($\mathcal{T}_2$). After the receivers $\mathcal{T}_1$ and $\mathcal{T}_3$ have merged their local graphs with the received graph, all transactional agents have up-to-date region serialization graphs.

5.3 Commit Processing

When a $\mathcal{T}_i$ wants to commit, it first has to check whether an incoming edge to $\mathcal{T}_i$'s node exists in its local replica of the region serialization graph (checking this with the

[3] To reduce the number of messages, it is possible to collect messages going to the same receiver. This does no effect the basic properties of the protocol, if eventually each message is sent to its receiver as described in this section.

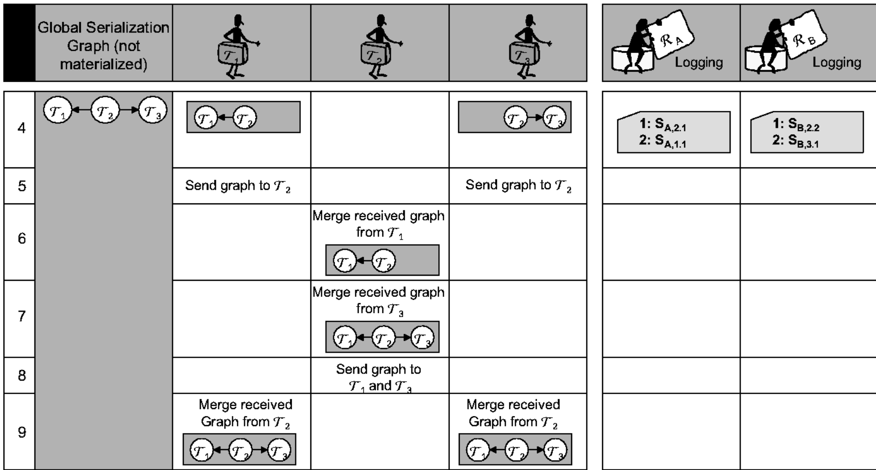

Fig. 5. Example: Messaging for keeping replica consistent

local copy is sufficient since all conflicts in which $\mathcal{T}_i$ is ordered after other transactional agents – i.e., all incoming edges to $\mathcal{T}_i$ – are immediately communicated to $\mathcal{T}_i$ after each service invocation). If this is the case, $\mathcal{T}_i$ has to wait until this edge disappears in order to correctly commit. Otherwise, $\mathcal{T}_i$ notifies all other $\mathcal{T}_j$ of its region to mark all edges it is involved in as removed and finally to remove the node corresponding to $\mathcal{T}_i$. $\mathcal{T}_i$ can be safely removed after its commit, since it does not have any incoming edge in the serialization graph. Then, this updated graph is also distributed by messages among all nodes having formed the region prior to the commit of $\mathcal{T}_i$. Hence, in certain cases, this might have the consequence that a region splits into two or even more independent regions.

5.4 Cycle Resolution

Due to the replication of metadata, each transactional agent of a region is able to locally check for cyclic conflicts. In case a cycle is detected, the associated transactional agents have to agree on one agent $\mathcal{T}_j$ to abort (this is orthogonal to the AMOR protocol and may be for instance the agent having caused the cycle or the one having invoked the least number of services so far). Then, the recovery process is started by $\mathcal{T}_j$ but might affect, due to cascading aborts, also other transactional agents. Due to the optimistic protocol where service invocations are executed (and committed) immediately although having side effects, cycle resolution may lead to a cascading abort of several active transactional agents.

5.5 Recovery

When a $\mathcal{T}_i$ aborts, it has to undo the effects of all regular services it has invoked so far in reverse order (or by invoking a service that jointly removes the effects of a set

	Global Serialization Graph (not materialized)	$\mathcal{T}_1$	$\mathcal{T}_2$	$\mathcal{T}_3$	$\mathcal{R}_A$ Logging	$\mathcal{R}_B$ Logging
10	$\mathcal{T}_1 \rightleftarrows \mathcal{T}_2 \rightarrow \mathcal{T}_3$		Invoke service on $\mathcal{R}_A$ Receive new conflict information $\mathcal{T}_1 \rightleftarrows \mathcal{T}_2 \rightarrow \mathcal{T}_3$		1: $S_{A,2.1}$ 2: $S_{A,1.1}$ 3: $S_{A,2.2}$	1: $S_{B,2.2}$ 2: $S_{B,3.1}$
11			Send graph to $\mathcal{T}_1$ and $\mathcal{T}_3$			
12		Merge received graph from $\mathcal{T}_2$ $\mathcal{T}_1 \rightleftarrows \mathcal{T}_2 \rightarrow \mathcal{T}_3$		Merge received graph from $\mathcal{T}_2$ $\mathcal{T}_1 \rightleftarrows \mathcal{T}_2 \rightarrow \mathcal{T}_3$		
13		Cycle detected. $\mathcal{T}_1$ has to roll back $\Rightarrow$ Send Undo Request for $S_{A,1.1}$ to $\mathcal{R}_A$				
14	$\mathcal{T}_1 \rightleftarrows \mathcal{T}_2 \rightarrow \mathcal{T}_3$				Receives message from $\mathcal{T}_1$ Determines the need to undo $S_{A,2.2}$ first	
15	$\mathcal{T}_1 \leftarrow \mathcal{T}_2 \rightarrow \mathcal{T}_3$		Receive message from $\mathcal{R}_A$, Undo $S_{A,2.2}$ on $\mathcal{R}_A$ $\mathcal{T}_1 \leftarrow \mathcal{T}_2 \rightarrow \mathcal{T}_3$ Send graph to $\mathcal{T}_1$ and $\mathcal{T}_3$		Compensation 1: $S_{A,2.1}$ 2: $S_{A,1.1}$ ~~3: $S_{A,2.2}$~~	
16	$\mathcal{T}_2 \rightarrow \mathcal{T}_3$	Merge received graph from $\mathcal{T}_2$, Undo $S_{A,1.1}$ on $\mathcal{R}_A$ and commit $\mathcal{T}_2 \rightarrow \mathcal{T}_3$		Merge received graph from $\mathcal{T}_2$ $\mathcal{T}_1 \leftarrow \mathcal{T}_2 \rightarrow \mathcal{T}_3$	Compensation 1: $S_{A,2.1}$ ~~2: $S_{A,1.1}$~~ ~~3: $S_{A,2.2}$~~	
17	$\mathcal{T}_2 \rightarrow \mathcal{T}_3$	X	Delete $\mathcal{T}_1$ $\mathcal{T}_2 \rightarrow \mathcal{T}_3$	Delete $\mathcal{T}_1$ $\mathcal{T}_2 \rightarrow \mathcal{T}_3$		

Fig. 6. Example: Cycle Resolution and Rollback

of regular service invocations). To do this correctly and efficiently, in a first step $\mathcal{T}_i$ sends a notification to all $\mathcal{R}_k$ where it has invoked regular services. In case no other conflicting service invocation $s_{k,j}$ of some $\mathcal{T}_j$ has occurred after $s_{k,i}$, the resource agent can immediately correctly undo the service. Otherwise, first all these $\mathcal{T}_j$ have to abort before $s_{k,i}^{-1}$ can be safely invoked by $\mathcal{T}_i$. To enforce this prerequisite, the corresponding abort request for $\mathcal{T}_j$ is generated by $\mathcal{R}_k$. This abort, in turn, may cascadingly lead to the abort of other transactional agents. In addition to determining whether other agents have to be aborted due to the abort request of $\mathcal{T}_i$, each $\mathcal{R}_k$ guarantees that it rejects service invocations which are in conflict with $s_{k,j}$, the service which effects are to be undone, until $s_{k,i}^{-1}$ has been invoked. When $\mathcal{R}_k$ is prepared (and when all cascading aborts are effected correctly), $\mathcal{T}_i$ is able to migrate to $\mathcal{R}_k$ and to invoke $s_{k,i}^{-1}$. As a consequence, all edges of the region serialization graph in which $s_{k,i}$ is involved are marked as removed. Finally – after all regular service invocations of $\mathcal{T}_i$ have been undone this way – $\mathcal{T}_i$ is able to commit.

We conclude our example by discussing the production, detection, and resolution of a cycle in the (region) serialization graph: In step 10, $\mathcal{T}_2$ invokes another service on $\mathcal{R}_A$. This invocation is assumed to be in conflict with $s_{A,1.1}$ invoked before by $\mathcal{T}_1$. So

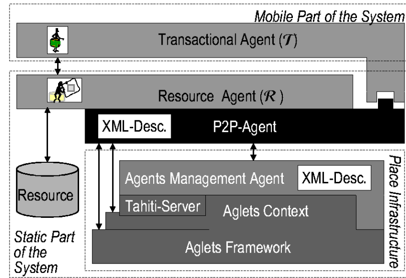

Fig. 7. Architecture of the AMOR Prototype

after updating the local serialization graph, a cycle is detected. Consequently, the new graph is sent to $\mathcal{T}_1$ and $\mathcal{T}_3$. Assume that $\mathcal{T}_1$ is chosen to rollback (step 13). Therefore, it first notifies $\mathcal{R}_A$. Since $\mathcal{T}_2$ has invoked service $s_{A,2.2}$ after $s_{A,1.1}$, the former has to be undone before $s^{-1}_{A,1.1}$ can be safely executed. Hence, $\mathcal{R}_A$ requests $\mathcal{T}_2$ to undo $s_{A,2.2}$ (step 14) before $s_{A,1.1}$ is undone (step 15). Since this is the only service $\mathcal{T}_1$ has invoked so far, $\mathcal{T}_1$ is committed, thereby leaving no effect to the system. After the commit of $\mathcal{T}_1$, it is removed from the region serialization graphs of $\mathcal{T}_2$ and $\mathcal{T}_3$, respectively (step 16). Then, the compensated service $s_{A,2.2}$ of $\mathcal{T}_2$ can again be invoked.

6 The AMOR-Prototype

6.1 Aglets System

We have implemented the AMOR prototype (Figure 7) on top of the Aglets framework [LO98]. This framework implements basic migration and communication functionality. The limitation of the basic Aglets framework is that an agent has to know the concrete address of other agents for means of communiaction, and, in case of migration, the address of the destination. This is realized in a run-time environment which is called Aglet Context. Each context can be understood as a peer in the network. On top of each peer, the framework provides a graphical user interface termed Tahiti Server. This GUI allows to easily start and stop agents resp. see which agents are currently running on a certain context.

6.2 Peer-to-Peer Agents

In a first step of our AMOR project, our aim was to overcome the restrictions of the Aglets System regarding migration and messaging, i.e., that the targets have to be hard-coded at build-time time. Instead, we have realized a higher level of abstraction called Peer-to-Peer Agents. These agents not only allow to find others with a certain name, they even allow to use predicates to specify agents (in case of messaging) and peers/places (in case of migration). Consequently, we equip the agents and places with descriptions. For example, the property 'type' is attached to agents allowing to differentiate for instance

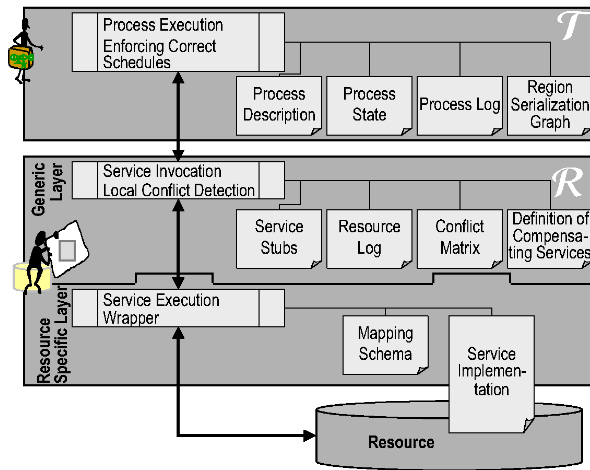

Fig. 8. Implementation Details of the AMOR Prototype

between agents providing weather forecasts (type='WeatherRA') and agents representing flight databases (type='FlightDatabaseRA').[4] Agents Management Agents (AMAs) bring these descriptions and the queries together [HS01]. They are transparent to the agent programmer, because they belong to the place infrastructure. On each place, exactly one AMA resides and works as a bookkeeper for the agents running locally. To connect a peer with the rest of the world, the AMA additionally manages the information about other peers. By this, the AMAs form a peer-to-peer network. This network is then used to propagate queries and evaluate them over the whole network or subparts of it. Thus, the P2P-Agents (together with the AMAs) realize a new mechanisms for migration and messaging on a higher level of abstraction.

6.3 Resource Agents

Resource Agents have two main tasks: i) they support transaction processing functionality and ii) they provide a uniform service interface. The second aspect deals with the problem of heterogeneity that emerges if we include existing resources into our system. This problem requires a solution which is tailored to the individual resource, whereas the first aspect can be realized in a general way.

This separation of resource-dependent and resource-independent aspects is reflected by a two layered implementation of the resource agent (Fig. 8). The upper, generic layer provides the interface to the transactional agents by offering the functionality to invoke services. Therefore, it has a general description of the services provided by a resource (service stubs). The service execution itself is implemented by the resource specific layer. This layer knows the mapping of the service stubs to the resource specific implementation of services.

[4] We assume a common terminology. By this, we can abstract in our work from the problem of defining and applying an appropriate ontology.

An important advantage of the traditional separation between interfaces and implementations of service stubs is that it allows to settle the local conflict detection and the compensation functionality in the generic layer. Thus, this can be used for any underlying resource by plugging the resource specific layer together with the generic one.

All resource agents implement the functionality to undo service invocations. Therefore, transactional agents contact the resource agents unless failure handling is not part of the process model in the form of alternative executions. The resource agents (in the generic layer) know for each service its inverse (compensating service). Whereas this allows the realization of (semantic) atomicity, the generic layer is also responsible for local conflict detection. Therefore, this generic layer incorporates a conflict matrix and logs the invoked services. This logging component uses a separate logging database on each place to ensure that the entries are made persistent.

6.4 Transactional Agents

Transactional Agents operate on top of these resource agents. They contain their process descriptions for example encompassing that we look for a flight from Zürich to Klagenfurt. Also, they contain the process state, e.g., that we have an offer for this flight for 299,- EUR has been discovered. But, additionally, each transactional agent also has to have a process log. This log memorizes which service has been invoked on which resource agent. If the transactional agent wants or is asked to rollback, it can determine which resource agents it has to contact to compensate its previously invoked services.

7 Related Work

The overall goal of the AMOR project is to provide a decentralized implementation for concurrency control and recovery. This is closely related to distributed deadlock detection [KKG99], which – together with a locking based protocol – can solve the same problem. However, such approaches are optimized for short-living transactions. Following an optimistic approach, AMOR also addresses long-running transactions.

For the same kind of environment, also timestamp ordering protocols [Tho79] can be applied, if global timestamps are available. In such approaches, serialization orders are not derived dynamically but are rather predefined by the timestamps assigned to each transaction. However, the risk of violating timestamp orders increases with the duration of transactions and finally leads to a large number of rollbacks induced by the violation of timestamp orders. Hence, also these protocols are rather suitable for short living transactions. Some approaches aim at weakening the strictness of the timestamp orders, e.g., dynamic timestamp allocation and validation [BEHR82], or multidimensional timestamp protocols [LB87]. These protocols dynamically assign timestamps. However, like all timestamp approaches, these protocols only address correct concurrency control and neglect recovery. Similar to traditional timestamp ordering, validation-based, optimistic concurrency control protocols such as BOCC [KR81] potentially come along with a large number of rollbacks when the duration of transactions and thus the number of conflicts increases. Essentially, these protocols check for the correctness of schedules not before commit time.

To support atomic applications in large-scale environments, the Transaction Internet Protocol (TIP) has been proposed. [LE98] Essentially, TIP implements a 2PC protocol. Recent extensions like Web Service Transactions (WS-Transactions) [Cea01] also support 'Business Activities', which are based on an optimistic service execution model with compensations. However, both, TIP and WS-Transactions, do not consider isolation.

Important related work regarding agents in general has been done by Chen and Dayal [CD00] and by Pitoura and Bhargava [PB95]. The latter provide an introductory discussion on using agents for accessing databases. The first discuss how multiple (non-mobile) agents can cooperate in order to guarantee atomic (but not necessarily isolated) workflow execution.

Focusing more on mobile agents, valuable work has been done in the context of transactions and workflow execution. The idea of mobile agents executing workflows – first published in [CGN96] – is similar to our approach to build transactional agent applications on top of services made available by different resource agents. Other approaches even address the intersection of mobile agent technology and transaction management. First results were achieved by enforcing atomicity and fault tolerance based on the concept of replication [SP98]. More recent approaches like [SAE01], albeit still concentrating on the atomicity aspect, also provide support for concurrency control. This is achieved by combining timestamp-ordering and 2PC [GR93], hence with a rather limited degree of concurrency. However, all these approaches aim not at combining optimistic transaction processing techniques with the isolation property. Therefore, they focus on short living transactions. Hence, they do not consider isolation and therefore lack support for reliable workflow resp. process execution in large, complex systems.

Finally, the underlying network model of the AMOR project is a peer-to-peer network. Recently, research in this direction has attracted quite some attention. This work was mainly driven by file sharing systems like Gnutella [Gnu]. Research concentrated very much on novel access methods, e.g. [Abe01]. However, such access methods are orthogonal to ARMOR and can therefore easily be integrated into our system.

8 Summary and Outlook

In this paper, we have presented the ideas of the AMOR project which allows for a novel decentralized implementation of transaction management in a peer-to-peer environment. Our approach is based on mobile transactional agents manipulating data provided by (non-mobile) resource agents. The resource agents are responsible for logging and (local) conflict detection. In contrast, it is the transactional agents' duty to ensure globally correct schedules by communication.

To this end, sophisticated replication management of metadata needed for synchronization purposes is applied. Each transactional agent receives information about local conflicts when it invokes a service. This information is used to update the local view on the relevant portion of global metadata, i.e., the region serialization graph. The different local replica of these graphs are kept consistent by means of communication at the agent level. To this end, we have introduced a new protocol that defines which information has to be transferred such that each transactional agent has sufficient information to decide whether a multi-agent execution is correct. For reasoning about correctness, we

apply the unified theory of concurrency control and recovery that jointly addresses the problems imposed by atomicity and isolation.

The two significant features of AMOR — the protocol allowing for decentralized transaction management, thus providing dedicated transactional execution guarantees for mobile agents and the support for location transparent migration and messaging — make the project a powerful effort towards transactional location transparent peer-to-peer information processing.

In our future work, we aim extending the existing AMOR framework by providing support for shielding unreliable network connections. To this end, we are working on a cost model by which the effects of disconnections and, more general, of different network latencies and bandwidths can be modeled. The goal is to extend the AMOR transaction model by these costs as dedicated quality-of-service parameters and to make use of this cost information for scheduling purposes. By this, we will build a sophisticated framework realizing our vision of reliable process execution in peer-to-peer networks.

References

[Abe01] K. Aberer. P-Grid: A Self-Organizing Access Structure for P2P Information Systems. In *9th Int. Conf. on Cooperative Information Systems, Trento, Italy*, 2001.

[Aea94] G. Alonso et al. Unifying Cconcurrency Control and Recovery of Transactions. In *Information Systems*, 1994.

[BEHR82] R. Bayer, K. Elhardt, J. Heigert, and A. Reiser. Dynamic Timestamp Allocation for Transactions in Database Systems. In *Proc. of the 2nd Int'l Symposium on Distributed Data Bases, Berlin*, 1982.

[BN97] P. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann, 1997.

[CD00] Q. Chen and U. Dayal. Multi-agent cooperative transactions for e-commerce. In *7th Int. Conference on Cooperative Information Systems, Eilat, Israel*, 2000.

[Cea01] F. Cabrera et al. Web services transaction, 2001. BEA Systems, IBM, Microsoft.

[CGN96] T. Cai, P. Gloor, and S. Nog. Dartflow: A Workflow Management System on the Web using Transportable Agents. Technical Report TR96-283, Dartmouth College, 1996.

[GM83] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems (TODS)*, 8(2):186–213, June 1983.

[Gnu] Gnutella. `http://www.gnutella.com`.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[HS01] K. Haller and H. Schuldt. Using Predicates for Specifying Targets of Migration and Messages in a Peer-to-Peer Mobile Agent Environment. In *5th Int. Conf. on Mobile Agents (MA), Atlanta, GA*, 2001.

[KKG99] N. Krivokapic, A. Kemper, and E. Gudes. Deadlock Detection in Distributed Database Systems: A New Algorithm and a Comparative Performance Analysis. *VLDB Journal*, 8(2):79–100, 1999.

[KR81] H. Kung and J. Robinson. On optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2), 1981.

[LB87] P. Leu and B. Bhargava. Multidimensional Timestamp Protocols for Concurrency Control. *IEEE Transactions on Software Engineering (TSE)*, 13(12):1238–1253, 1987.

[LE98] J. Lyon L. Evans, J. Klein. Transaction Internet Protocol Version 3.0. http://www.ietf.org/rfc/rfc2371.txt, 1998. IETF RFC 2371.

[LO98] D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley Longman, 1998.

[PB95] E. Pitoura and B. Bhargava. A framework for providing consistent and recoverable agent-based access to heterogeneous mobile databases. *SIGMOD Record*, 24(3):44–49, 1995.

[SABS02] H. Schuldt, G. Alonso, C. Beeri, and H.-J. Schek. Atomicity and isolation for transactional processes. *ACM Transactions on Database Systems (TODS)*, 27(1), 2002.

[SAE01] R. Sher, Y. Aridor, and O. Etzion. Mobile Transactional Agents. In *21st Int. Conf. on Distributed Computing Systems, Phoenix, AZ*, 2001.

[SP98] A. Silva and R. Popescu-Zeletin. An Approach for Providing Mobile Agent Fault Tolerance. In *2d Int. Workshop on Mobile Agents, Stuttgart, Germany*, 1998.

[Tay86] D. Taylor. Concurrency and Forward Recovery in Atomic Actions. *IEEE Transactions on Software Engineering*, SE-12(1):69–78, January 1986.

[Tho79] R. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems (TODS)*, 4(2):180–209, June 1979.

[VHBS98] R. Vingralek, H. Hasse-Ye, Y. Breitbart, and H.-J. Schek. Unifying Concurrency Control and Recovery of Transactions with Semantically Rich Operations. *Theoretical Computer Science*, 190(2), 1998.

[Whi97] J. E. White. Telescript. In W. Cockayne and M. Zyda, editors, *Mobile Agents: Explanations and Examples*, 1997.