In: Proceedings of the TES'01 Workshop on Technologies for E-Services.
Rome, Italy, September 14-15, 2001.

Supporting Reliable Transactional Business Processes by Publish/Subscribe Techniques*

Christoph Schuler, Heiko Schuldt, and Hans–Jörg Schek

Institute of Information Systems
Swiss Federal Institute of Technology (ETH)
ETH Zentrum
8092 Zürich, Switzerland
Email: {schuler,schuldt,schek}@inf.ethz.ch

Abstract. Processes have increasingly become an important design principle for complex intra- and inter-organizational e-services. In particular, processes allow to provide value-added services by seamlessly combining existing e-services into a coherent whole, even across corporate boundaries. Process management approaches support the definition and the execution of predefined processes as distributed applications. They ensure that execution guarantees are observed even in the presence of failures and concurrency. The implementation of a process management execution environment is a challenging task in several aspects. First, the processes to be executed are not necessarily static and follow a predefined pattern but must be generated dynamically (e.g., choosing the best offer in a pre-sales interaction). Second, deferring the execution of some application services in case of overload or unavailability is often not acceptable and must be avoided by exploiting replicated services or even by automatically adding such services, and by monitoring and balancing the load. Third, in order to avoid a bottleneck at the process coordinator level, a centralized implementation must be avoided as much as possible. Hence, a framework is needed which supports both the modularization of the process coordinator's functionality and the flexibility needed for dynamically generating and adopting processes. In this paper we show how publish/subscribe techniques can be used for the implementation of process management. We show how the overall architecture looks like when using a computer cluster and publish/subscribe components as the basic infrastructure to drive the enactment of processes. In particular we describe how load balancing, process navigation, failure handling, and process monitoring is supported with minimal intervention of a centralized coordinator.

* Part of this work has been funded by the Swiss National Science Foundation under the project INVENT.

1 Introduction

E-services are, in general, complex sequences of individual steps needed to achieve some business task. They are not necessarily restricted to be executed within a single system but may rather be distributed, both physically and organizationally. Such complex e-services can be found in b2c (business-to-customer) as well as in b2b (business-to-business) interactions, thereby even allowing to cross corporate boundaries by combining existing e-services.

Processes have increasingly become an important design principle for complex intra- and inter-organizational e-services since they seamlessly allow to encompass the individual constraints that can be found in e-services by means of control and data flow dependencies between single process steps. Most existing process coordinators require that the individual steps of a process have to be defined at built-time. E-services being characterized by such static processes can be found, for instance, in e-commerce payment interactions [3, 15]. However, a considerable class of e-services is characterized by the lack of a pre-defined structure [12], e.g., pre-sales interactions.

Consider, as an example for the management of processes which are dynamically generated at run-time, the supply chain of a car manufacturing company United Cars (UC). UC uses an automated management system to manage parts stored on stock. In case this system detects that the number of screws of a particular type, say *SG-H 37 ZC∗*, has fallen below a certain threshold, a process ordering new screws has to be triggered. In order to optimize costs, standard parts like screws are dealt with a fixed, single supplier, but they are rather ordered from the supplier offering the best price currently available on the market. To this end, UC first has to gather current offers for 10'000 screws of type *SG-H 37 ZC∗* from all potential dealers. According to the results of this phase, an order can be placed by the purchasing department. The purchase order will be performed and the screws of the type *SG-H 37 ZC∗* will be inserted into stock. In order to guarantee that the production of UC cars will never be stopped due to some missing screws, the order process has to terminate correctly and as fast as possible by using clustered services, hence needs dedicated execution guarantees.

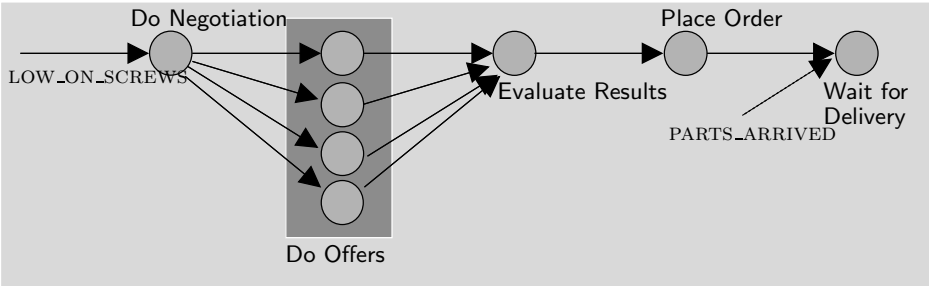

Fig. 1. *Sample Process: Automated order processing for parts stored on stock*

While the example depicted in Figure 1 focuses on execution guarantees and dynamic assignment of services, other applications may introduce additional requirements. Consider, for instance, business processes in the area of multimedia object management which gather image data and finally maintain a search engine on them [20]. These processes feature a high need of replicated services in order to speed up execution. To support this, the running components have to be monitored in oder to optimize load balancing. Moreover, this application shows the need of sophisticated resource management at coordinator level: e.g., processing 100'000 images and inserting them into an index will result in 100'000 processes running at the same time which leads to a potential bottleneck of a centralized process manager. Therefore, we see a need to decouple the functionality of a process coordinator and to distribute the individual run-time services to different components, thereby minimizing the intervention of a centralized coordinator and avoiding a bottleneck at coordinator level. In summing up, in order to achieve execution guarantees we must monitor the availability and the load of services. Based on monitoring we must dynamically select the best available service. The implementation must avoid a centralized implementation of the process manager whenever possible.

In this situation — and this is the contribution of the paper — we have come up with a solution for the implementation of a sophisticated process manager that combines the advantages of publish/subscribe techniques with transactional process management. The key feature we are using repeatedly is the following: Systems providing certain services can register (*subscribe*) for the service they offer. Whenever an event like, for instance, the termination of a process step is evoked (*publish*), the process coordinator is able to dynamically chose the next service among all that have been subscribed, thereby considering only those services that are currently available. Based on this publish/subscribe communication infrastructure, many features like process navigation, failure handling, load balancing, and monitoring can be implemented in an elegant and flexible way.

The paper is organized as follows: in Section 2, we briefly introduce the process model we are relying on while Section 3 introduces the idea of publish/subscribe techniques. The application of publish/subscribe techniques to process management is presented in Section 4. Section 5 discusses related work and Section 6 concludes.

2 Transactional Processes

In this section, we introduce the basic ideas of transactional processes which will form the theoretical foundation for the application of publish/subscribe techniques to implementing complex e-services.

2.1 Process Model

A process is a partially ordered collection of activities. In particular, processes introduce flow of control and flow of data between activities as basic semantic

elements. Activities, in turn, correspond to invocations of application services. In order to take into account that certain steps within a process are irreversible, activities can be characterized in terms of their termination guarantees: they are either *compensatable, retriable*, or *pivot* — according to the flex transactions model [9, 21]. Compensatable activities can be semantically undone after they have committed, pivot activities are those which are not compensatable (when no appropriate compensation is available or when compensation is too expensive and thus has to be avoided), and retriable activities are the ones that are guaranteed to terminate successfully. Due to the special characteristics of processes which are in general long-running and complex, it is not feasible to encompass all activities of a process within a single distributed transaction. Each activity is required to commit immediately after it has been completed — which is even required for pivots. Hence, additional effort is required to handle failures correctly within processes. To this end, the regular order between activities (the *precedence order*) is complemented by an additional order, the *preference order*, indicating alternative executions that can be taken in case of failures [18].

Based on the different termination properties of activities and the precedence and preference orders, it can be validated whether a single process is defined correctly. This is the case when all possible failures of process activities can be handled correctly by either undoing all completed activities (when only compensatable activities have committed) or by executing a safe alternative consisting only of retriable activities (thereby, also failures can be handled which occur after a pivot activity has been committed). Processes for which these structural constraints hold are called *processes with guaranteed termination* [18]. This inherent correctness property of transactional processes is an important and powerful generalization of the "all-or-nothing" semantics of traditional ACID transactions since it ensures that one of eventually many valid executions (specified by alternatives) is effected, thereby ensuring that the system in a consistent state after process completion.

2.2 Process Execution

The execution of transactional processes is controlled by a process coordinator. Starting with the correct specification of single processes having guaranteed termination property, the process coordinator's task is to enforce the correct execution of transactional processes even in the presence of failures and concurrency, i.e., when different processes simultaneously access shared resources. The key aspects of the transactional process coordinator can briefly be summarized as follows: it acts as a kind of transaction scheduler that is more general than a traditional database scheduler in that it

i.) knows about semantic commutativity of activities,
ii.) knows about the termination properties of activities,
iii.) exploits the regular precedence order of processes when executing activities and knows about alternative executions paths in case of failures, and
iv.) optimizes execution costs of processes by choosing the best alternative among the set of alternatives specified by the preference order [17] at each state.

3 Publish/Subscribe Techniques

In this section, we introduce the basic concepts of the publish/subscribe paradigm and give a brief overview of the different applications of this technique.

3.1 Introduction to Publish/Subscribe

Message-oriented middleware (MOM) loosely couples individual systems by replacing the commonly exploited synchronous, RPC-like invocations with the asynchronous transfer of messages. The publish/subscribe (pub/sub) paradigm is a special form of MOM which allows to further decouple a sender and the receiver(s) of messages [14]. The key characteristics of pub/sub interactions is an additional indirection in the communication between sender and receiver(s): rather than addressing a message directly, a sender associates it with a certain topic, i.e., a description of the message content, and does not have to have any information about who will be the recipient of this particular message. Yet, *publishing* a message just requires transferring it to a dedicated *message broker*. The latter is then responsible for distributing this message, according to meta data indicating who has previously shown interest of messages of that particular topic. The procedure of registering an individual client profile of topics with the message broker is referred to as *subscription*. An alternative to the description of messages by means of predefined topics is to use filter predicates specified by recipients to analyze whether or not publications are of particular interest (consider, for instance, an electronic car auction where a client is interested in any offer of a VW New Beetle for less than $10.000). Filter predicates increase flexibility by allowing arbitrary publications, thereby avoiding the restriction of the publisher to the usage of a common topics schema. Moreover, when given the capability to persistently store messages together with the associated description, the message broker may even allow to distribute messages to receivers which did not exist at the time the message has been published. Persistent queuing functionality also allows to guarantee correct message transfer. A message is first inserted into the publisher's local queue. The publication, which then corresponds to the transfer from this local queue to the queue of incoming messages located at the broker's site, is implemented as a two-phase commit (2PC) [6] coordinated distributed transaction. In a similar way, also the transfer of messages from the broker to the subscriber is treated, yet in an independent transaction, thus achieving asynchronous publisher/subscriber interactions. An additional degree of freedom in pub/sub interactions, depending on the individual semantics of a concrete application, is whether or not it is sufficient to forward a message to exactly one subscriber, to a certain set of subscribers, or to all of them.

While most commercial pub/sub implementations (e.g., IBM's MQSeries [10] or the implementations of the CORBA Event Service [13]) follow the broker approach, certain products even avoid the centralized broker. In TIB/Rendezvous [19] of TIBCO, for instance, a publisher distributes messages via broadcast to

all potential subscribers which then have to filter locally the messages they are interested in.

3.2 Applications of Pub/Sub Techniques

The pub/sub idea was originally introduced in the context of mailing lists for Internet newsgroups where a customer explicitly has to subscribe for a couple of predefined topics so as to receive an e-mail whenever a message to one of these topics is published. A generalization of this first application can be found in systems where the recipients of messages are no longer human beings but arbitrary application programs (e.g., electronic stock brokers logging each significant change in the stock value of the shares of United Cars). The usage of pub/sub techniques can even be further generalized, leading to loosely coupled distributed information systems, when the receiver actively reacts on a message. The rationale behind this approach is that a message corresponds to an event that has been evoked by some application (the publisher) and requires certain response by another program (the subscriber).

4 Processes Support by Pub/Sub Technologies

Our PM^{PS} approach relies on the above mentioned idea of loosely coupled distributed information systems and addresses the event-driven execution of business processes (process management, PM) by pub/sub (PS) techniques. By this, core system functionality can be mapped to the underlying pub/sub infrastructure without having to deal with dedicated components for various runtime services such as, for instance, distribution of load information for load balancing purposes. Activities of a process correspond to services provided by one component of the system (in certain cases, services are replicated and thus can be provided by different component systems). In order to avoid that components drive the execution of processes in a bilateral way by explicitly invoking the subsequent service at another host by remote procedure call once a local service has terminated, pub/sub techniques are used. To this end, components offering services which are used within processes have to register the service they offer (subscription). After an event is published —via an appropriate message, i.e, the completion of some task— that triggers another task, the service corresponding to the latter is invoked. Consider, for example, a component C_1 offering a service s_1 which corresponds to activity a' of some process P. C_1 can subscribe this service for events *"*s_1*.start"* which, in turn, are generated after the event *"*s_0*.terminated"* has been raised where s_0 is a service corresponding to an activity a^* that directly precedes a' in process P with respect to the precedence order. Hence, the component S_0 executing the service corresponding to activity A_0 does not have to be aware of where process execution will be continued; server S_1 is contacted by the PM^{PS} system rather than by S_0. While control flow is initiated by means of messages, data to be shipped between tasks is also encompassed within these messages. A crucial aspect for driving the execution of

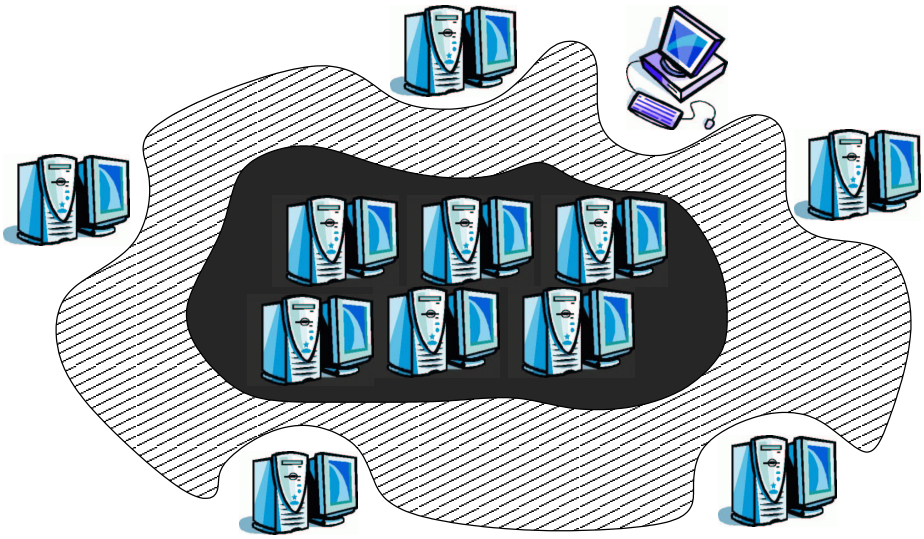

Fig. 2. *Different types of components: KER for process execution (inner part) and APPS for individual process steps (outer part)*

processes in a reliable way is to exploit persistent queuing mechanisms to avoid the loss of messages.

We use pub/sub techniques also to distribute the implementation of our process coordinator. This helps to improve load balancing, process navigation, failure handling, and monitoring. The implementation of these features are discussed in the following subsections.

4.1 System Model

The PM^{PS} approach is designed to run in a heterogeneous and distributed environment. We distinguish two types of components participating in the system. The first type, called *KER* consisting of a set of dedicated servers, builds the PM^{PS} kernel (depicted in Figure 2 as the machines belonging to the inner, dark-shaded area). These components have to feature high availability since they are exploited to drive the execution of transactional processes and to add services at run-time, e.g., providing load balancing, or monitoring functionality for the states of processes. The second type, called *APPS*, is made up of the components providing application services which will be combined by processes (depicted in the outer part of Figure 2). Hence, these components are used for the execution of single process steps rather than to control the actual process execution. In particular, these components can dynamically change the set of services they provide, and they can themselves dynamically join and leave the system. The overall system should not be affected, even if *APPS* components crash and never rejoin the system.

Since all communication in the system is based on asynchronous messages and pub/sub techniques, no *APPS* component has to be aware of the network address of any another component. The communication layer connecting all components of either type just has to know how to contact the central pub/sub directory which is hosted at one of the *KER* components (and which is, according to the requirements imposed for these components, highly available and/or redundant). This directory lookup service maintains information on pub/sub topics and all subscriptions to these topics. In the PM^{PS} approach to process management, topics coincide both with the different events that are raised by *APPS* components so as to signal the state of service invocations (whether they are successfully completed or whether they have failed) and with the events generated by the *KER* components, reflecting relevant changes in the meta data of the system.

Whenever a new component of the *APPS* type joins the system (note that all components of the *KER* type are static and thus can neither dynamically join nor leave the system), it registers itself at the pub/sub directory by specifying the different services it provides.

4.2 Load Balancing

In order to the increase the performance and availability of *APPS* components, services can be configured to run on more than one component. In this case, the pub/sub run-time infrastructure has to provide a special component implementing a dedicated load balancing service. This service routes incoming message to exactly one subscriber of a subscribed group, where the service corresponding to that particular message can be executed. Since the load balancing component can decide freely —but based on load information it dynamically gathers— where to route a specific task, the result of an execution has to be independent of the location where the service is executed, i.e., the services among which the load balancing component chooses have to be (semantically) identical.

To allow for the implementation of sophisticated load balancing services, each *APPS* component has to publish its actual load. To prevent a high system load, due to heavy load update information, PM^{PS} implements an event-based mechanism that notifies the load balancing service whenever significant changes in the local load occur (i.e., when the deviation of the load exceeds a pre-defined threshold). Hence, the load balancing service acts as a subscriber for load information published by the *APPS* components.

4.3 Process Navigation

In addition to transparent service invocation provided by the meta information maintained at the pub/sub directory allowing to link two subsequent steps of a process, the overall navigation within processes has to take place on top of this infrastructure.

Therefore, a process description lookup service must be hosted at a *KER* component. This service maintains information about all defined processes, following the model of transactional processes (c.f. Section 2.1). For process modeling purposes, we are using a graphical process modeling and simulation tool, IvyFrame of IvyTeam [8], which we have extended in PM^{PS} in order to export process models in the format used by the process description lookup service (i.e., in the form of pairs of consecutive activities and the corresponding services that have to be invoked to execute these activities). In particular, the process description lookup service has to map events raised by some *APPS* component like $s_0.terminated$ to events that trigger control flow and that start the subsequent activity by invoking its associated service: $s_1.start$. To this end, the process description lookup service needs both information on the correlation between process activities and the associated services as well as on the control flow of a process (which includes both the precedence order for regular execution and the preference order for alternative execution which have to be effected in case of failure).

In general, this mapping is realized by a centralized service which is a subscriber for all $?.terminated$ and $?.failed$ events, i.e., events that signal the successful completion or failure of services. But this navigation by mapping events can also be de–centralized: in this case, the local pub/sub daemon which anyway resides on each *APPS* component to provide transparent communication by means of pub/sub , i.e., to publish events after a service has terminated, takes over this task. To do this, the daemon has to locally cache global information maintained by the process description lookup service. Based on this cached information, the daemon can decide locally which event to publish after the termination of a service. The prerequisite for this is, however, that the local cache is kept up to date such that dynamic changes in the process description are immediately propagated to all local daemons. Again, this is realized via pub/sub techniques in that each daemon subscribes for process description lookup data which is published by the centralized service whenever changes are performed. This allows to distribute meta data of the process lookup service for cache update purposes.

4.4 Failure Handling

Following the model of transactional processes, the failures of single activities are handled by either re–invoking the service (retry), executing alternatives, or by compensation. In all cases, the strategy is present in the control flow of the process (preference and precedence order, respectively) and the termination characteristics of the activity. To this end, the failure of some local service s_1 corresponding to an activity A_1 is published *"$s_1.failed$"* and can be handled by the process navigation similar to the successful treatment (However, in this case compensation is required or alternative execution are effected rather than continuing with the regular control flow).

The failure of individual *APPS* components, however, requires additional effort. In particular, it must be guaranteed that all events that have been pub-

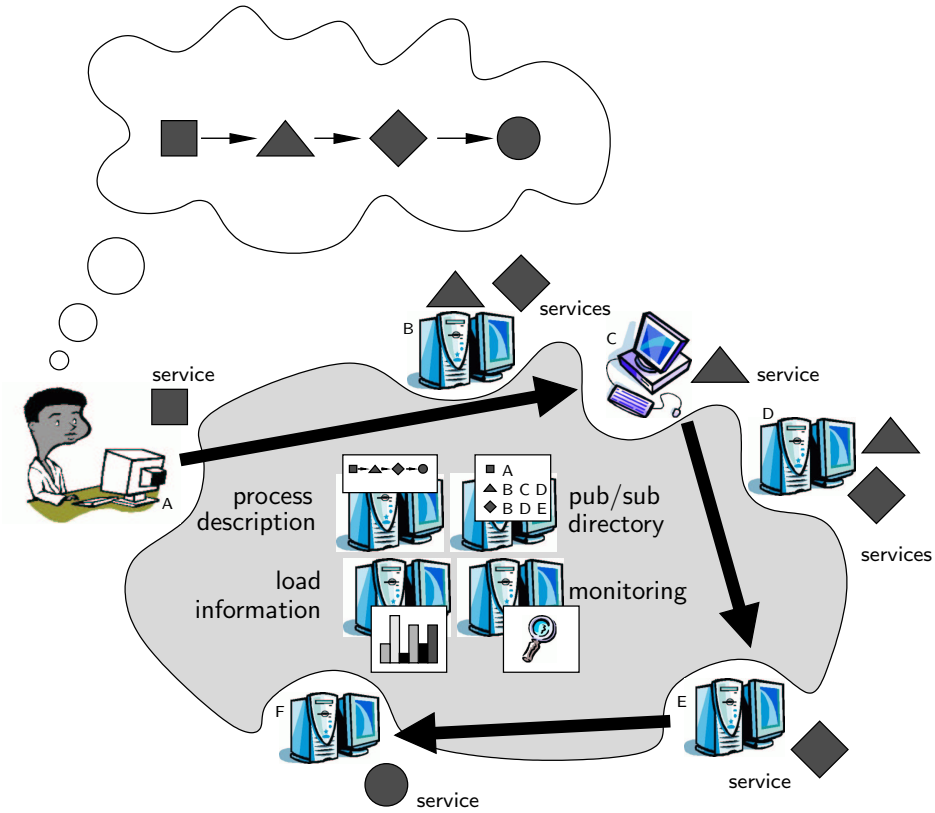

Fig. 3. *Dynamic process execution by the PM^{PS} system*

lished will be processed by some subscriber. Hence, appropriate support by the underlying pub/sub infrastructure of PM^{PS} is required. Essentially, following the ideas of queued transactions [2], persistent queuing technology can be applied in order to guarantee not only that a message corresponding to an event arrives at the subscriber but also that it is actually processed there by some service s (before a new event signaling the state of s is raised). To this end, the delivery of a publication which corresponds to a dequeue operation from a persistent queue maintained by the PM^{PS} system, the service itself, and the enqueue of the event signaling the state of the service (success or failure) have to be encompassed within a single transaction. This is the task of the local daemon which not only provides the basic pub/sub communication facilities but which also relates the invocation of local services to the events being published and consumed.

In case some *APPS* component having subscribed for the execution of a service cannot be reached, three corrective strategies exist:

i.) if there are other components having subscribed for that particular service, they can be chosen instead (in an order imposed by their current load), or
ii.) if alternative services are defined in the process (by means of preference orders, c.f. 2.1), these services can be invoked, or
iii.) in case a particular service is not replicated and no alternative is defined, process execution has to be frozen until either that component is available or until another component has subscribed for this service.

4.5 Monitoring

Meta data on the state of the overall system is very important, both in terms of individual components (what is their current load, what services they are currently executing, etc.) and in terms of individual processes (what is their current state). To this end, an additional run-time service, *monitoring*, located at one of the *KER* components, gathers all information about components connected to the system, active processes, and so forth. By subscribing itself as consumer of all types of system state messages, the monitoring component is seamlessly kept up-to-date without requiring any interception mechanisms on the invocation of services. By using appropriate filter predicates associated with the subscription, the granularity of monitoring can be tailored to the individual needs of processes or users. When, for instance, the state of particular process P_1 has to be monitored, only events corresponding to the enactment of this process have to be gathered, i.e., subscription is restricted to $P_1.?$.

In order to graphically depict the state of processes in a user-friendly way, PM^{PS} exploits IvyFrame, the same tool that is already used for process modeling purposes. Hence, there is a unique interface towards the users of the PM^{PS} system.

4.6 Application of Pub/Sub Techniques: Example Revisited

Coming back to our initial example, the supply chain management of United Cars, a message *low_on_parts* is published after the stock management system detects the need of screws. This message contains information about the exact type of the parts needed: (screws, *SG-H 37 ZC*∗).

After the message is published, the pub/sub directory process description service checks whether subscriptions to this message exist. In this case, it is detected that the message corresponds to an *order_parts* process —according to the process description stored in the repository— which has to be executed by copying information from the *low_on_parts* message to the context of the new process. In order to execute the first activity, it sends a message with the topic *do_negotiation.start* to the system. The system routes this message to the *APPS* component offering this service and having subscribed for this particular topic before. In this case, subscriptions consist of suppliers announcing the availability of screws in their catalog. In order to process the initial message published by UC (which corresponds to a call for bids), each subscribed supplier is contacted. The implementation of the *do_negotiation* service uses the pub/sub infrastructure to

publish a *get_offers* to reach all suppliers currently connected to the system. The services executed at the supplier sites generate bids for the number and type of screws requested by UC. In addition to the individual bids, they send back the service name and execution parameters specification (which would be required when the bid is chosen for the execution of the ordering service). According to this information, a decision can be made about which dealer will be selected.

After the bidding step is completed, the *do_negotiation* service terminates which results in a *do_negotiation.terminated* message. First, this message is mapped to the start message of the next step with respect to the control flow (*place_order.start*). Second, this message is also consumed by additional *KER* components, for instance for monitoring purposes.

The "place order service" compiles the information for the order and calls the order service at the dealers place. After the execution of this purchase order, the next step in the process is an activity that waits for the message indicating that the parts are shipped and inserted into the stock. In case that this does not take place within a given time–frame, a timeout occurs and an alternative branch of the process is executed so as to perform additional activities like the sending of reminder or the cancellation of the order and the placement of an order using a different supplier. As a last activity, the payment will take place.

Using transactional processes, it is guaranteed that all possible failures of services are handled correctly and the system always terminates in a consistent state. In this case, whenever the screw ordering process is terminated, the stock is refilled with *SG-H 37 ZC∗* screws, either stemming from the initially contacted supplier or from the supplier chosen alternatively in case the initial one could not proceed its order.

5 Related Work

There are many different approaches to realize dynamic process management systems. In general, there are two possibilities to handle dynamic changes during process execution. First, this can be achieved by changing the process definition at run–time. This requires a migration of running instances to the new schema. $ADEPT_{FLEX}$ [16] and rule–based approaches as they can be found, for instance, in HematoWork [11], handle this migration while preserving consistency of the process instances.

A second method to deal with dynamic aspects is to define a static process definition, but assign services at run–time and therefore achieve dynamic process execution. CrossFlow [7] and *e*Flow [4] focus on negotiation and service discovery. CrossFlow [7] uses an electronic market to support the dynamic assignment of services by advertising and searching for compatible business partners. Other approaches like *ADEPT* [1] realize a similar effect by dynamically assigning process managers. The *e*Flow [5, 4] approach handles changes on process programs as well as on process instances.

The PM^{PS} approach uses a static process program and dynamically resolves available *APPS* servers to decide where to execute activities, while focusing on

execution guarantees, using a transparent communication layer to describe and to invoke services by pub/sub.

Commercially available products like IBM's MQSeries Workflows [10] are using persistent queues in order to call activities on remote systems. The MQSeries family also includes publish/subscribe functionality but with a slightly different focus than PM^{PS}, namely to loosely couple applications rather than controlling the execution of processes.

All previously discussed approaches rely on a central process coordinator, while PM^{PS} decouples this functionality in order to distribute process navigation as well as other run–time services.

6 Conclusion

Transactional process management can be used to integrate existing e–services seamlessly into new business processes by plugging existing components together.

This paper has shown how to realize a distributed implementation of a process management system by dividing the functionality into a bunch of decoupled services. Focused on process navigation, failure handling, load balancing, and monitoring, we have described how to map process management to the pub/sub communication primitives.

Our modular framework, supporting transactional process coordination as well as dynamically adapting the execution of processes, can deal with the different particular problems of e-commerce scenarios. Steps of a process program are not hard–wired to application servers. By this, processes can even be executed in a highly dynamic environment. The PM^{PS} system determines the server of a certain service at run–time, so that availability as well as load balancing is taken into account. Yet, the deferment of process execution as a result of unavailable or overloaded components can be avoided.

Using the ideas of transactional processes, the consistent termination of every business process is guaranteed, such that key transactional properties are fulfilled.

As future work, a concurrency control service will be integrated into the system by using a second indirection in the control flow. The goal is that exploiting a locking protocol for processes, e.g., [17] at process coordinator level, allows for the correct parallelization of concurrent processes can be guaranteed, by seamlessly intercepting service calls within the pub/sub infrastructure.

References

1. Th. Bauer and P. Dadam. Efficient Distributed Workflow Management Based on Variable Server Assignments. In *Proceedings 12th Conference on Advanced Information Systems Engineering*, pages 94–109, Stockholm, Sweden, S., June 2000.
2. P. Bernstein and E. Newcomer. *Principles of Transaction Processing.* Morgan Kaufmann Publishers, 1997.

3. J. Camp, M. Sirbu, and D. Tygar. Token and Notational Money in Electronic Commerce. In *Proceedings of the 1st USENIX Workshop on Electronic Commerce*, pages 1–12, July 1995.
4. F. Casati, U. Dayal, and M. Shan. E–Business Applications for Supply Chain Automation: Challenges. In *Proceedings of the 17th International Conference on Data Engineering*, Heidelberg, Germany, April 2001.
5. F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M. Shan. Adaptive and Dynamic Service Composition in eFlow. In *Proceedings 12th Conference on Advanced Information Systems Engineering*, Stockholm, Sweden, S., June 2000.
6. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.
7. P. Grefen, K. Aberer, H. Ludwig, and Y. Hoffner. CrossFlow: Cross–Organizational Workflow Management for Service Outsourcing in Dynamic Virtual Enterprises. *IEEE Data Engineering Bulletin*, 24:52–57, 2001.
8. IvyTeam, Zug, Switzerland. `http://www.ivyteam.com`.
9. S. Mehrotra, R. Rastogi, A. Silberschatz, and H. Korth. A Transaction Model for Multidatabase Systems. In *Proceedings of the 12th International Conference on Distributed Computing Systems (ICDCS'92)*, pages 56–63, Yokohama, Japan, June 1992. IEEE Computer Society Press.
10. MQSeries Publish/Subscribe User's Guide. IBM Red Book, No. GC34-5269-05, 2000. IBM, International Business Machines Corporation.
11. R. Müller and E. Rahm. Rule-Based Dynamic Modification of Workflows in a Medical Domain. In *Proceedings of Datenbanksysteme in Büro, Technik und Wissenschaft (BTW'99)*, Informatik Aktuell, pages 429–448, Freiburg, Germany, March 1999. Springer Verlag.
12. P. Muth, J. Weissenfels, and G. Weikum. What Workflow Technology can do for Electronic Commerce. In *Proceedings of the EURO-MED NET Conference*, Nicosia, Cyprus, March 1998.
13. OMG. Object Manegement Group. `http://www.omg.org`.
14. R. Orfali, D. Harkey, and J. Edwards. *Client/Server Survival Guide*. John Wiley & Sons, 3rd edition, 1999.
15. A. Popovici, H. Schuldt, and H.-J. Schek. Generation and Verification of Heterogeneous Purchase Processes. In *Proceedings of the International Workshop on Technologies for E–Services (TES'00)*, Cairo, Egypt, September 2000.
16. M. Reichert and P. Dadam. $ADEPT_{flex}$ – Supporting Dynamic Changes of Workflows Without Losing Control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.
17. H. Schuldt. Process Locking: A Protocol based on Ordered Shared Locks for the Execution of Transactional Processes. In *Proceedings of the 20th ACM Symposium on Principles of Database Systems (PODS'01)*, Santa Barbara, California, USA, May 2001. ACM Press.
18. H. Schuldt, G. Alonso, and H.-J. Schek. Concurrency Control and Recovery in Transactional Process Management. In *Proceedings of the 18th ACM Symposium on Principles of Database Systems (PODS'99)*, pages 316–326, Philadelphia, Pennsylvania, USA, May/June 1999. ACM Press.
19. TIB/Rendezvous. White Paper, 1999. TIBCO Software Inc.
20. Roger Weber and Hans-J. Schek. A distributed image-database architecture for efficient insertion and retrieval. In *Fifth International Workshop on Multimedia Information Systems (MIS'99)*, Indian Wells, Palm Springs Desert, California, October 21–23 1999.

21. A. Zhang, M. Nodine, B. Bhargava, and O. Bukhres. Ensuring Relaxed Atomicity for Flexible Transactions in Multidatabase Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'94)*, pages 67–78, Minneapolis, Minnesota, USA, May 1994. ACM Press.