# Ontology-Driven Transaction Monitor for Mobile Services

Vagan Terziyan

Faculty of Information Technology, University of Jyvaskyla, P.O. Box 35, FIN-40351, Jyvaskyla, Finland, e-mail: vagan@it.jyu.fi

## Abstract

The main goals of this paper are to provide the description of ontology-driven Transaction Monitor (TM) for mobile business applications and its implementation issues. The approach is based on assumption that TM is an independent mobile terminal application, which can integrate different distributed external e-services by managing appropriate transactional processes. We use the ontology-based framework for transaction management so that the TM will be able to manage transaction across multiple e-services and we consider management of distributed location-based services as an example of such ontology-based TM implementation. The core of the approach is ontologies. Ontologies should be "placed" both in mobile terminals and in e-services. They define common multiple clients - multiple services standards and vocabularies for the use of the names, types, schemas, default values for parameters, atomic service actions with appropriate structure of their input and output. In our implementation ontologies help to the TM to deal with multiple services during transactions and simplify appropriate user interface.

## 1 Introduction

M-commerce refers to e-commerce activities relying on mobile e-commerce transactions.

*A mobile e-commerce transaction is any type of business transaction of an economic value that is conducted using a mobile terminal that communicates over a wireless telecommunications or Personal Area Network with the e-commerce infrastructure*.

Transaction management was first developed in the database management context. In the modern sense the requirements were set up in the context of relational database management systems. The main targets of transaction management, concurrency control of the simultaneously running applications and recovery in the case of crashes, are taken care of automatically by the database management system. The executions should exhibit "ACID" properties. **A**tomicity means that either all the retrievals, updates and deletions within data are performed or all of its effects are aborted. **C**onsistency means that the transaction program is semantically correct, i.e. it keeps the database in a consistent state if run alone. **I**solation means that the concurrent executions of different transaction programs should behave towards the user and database as if there were no concurrency in running them. **D**urability means that the results of successfully terminated transaction program executions must persist in the state of database irrespective of the other concurrently running transactions or system crashes that might destroy the state of the system produced by the transaction. The results of the earlier studies on transaction management issues are exhaustively represented in [1] and [7].

There are some differences between e- and m-commerce transactions that should be taken into account in transaction management [12]:

- the mobile e-commerce environment is hostile in the sense that the customers or merchants might be traitorous;
- in the mobile e-commerce environment a terminal can easily be stolen and taken in an unauthorised use;
- the terminals in the mobile e-commerce have much less processor, memory and other resources;
- the security mechanisms are different ;
- existing m-commerce infrastructure is not homogeneous, but rather different from country to country;
- "location invalidation", i.e., transaction becoming invalid due to out-dated location information is specific to wireless.

Mobile E-commerce transactions are currently being developed in an industry-led consortium called MeT-forum [5]. The work has produced a public white paper [4] where the opportunities and risks of m-commerce are discussed. Scenarios (business models) for the m-commerce are currently being developed.

Location-based services (LBS) are based on the fact that the terminal has a position on the earth and this is made known to applications running on the infrastructure. The infrastructure can be running on mobile operator's sphere of control or on some external service provider. A typical query is: "Where am I now?", "Where is the cheapest restaurant that is 500 m away?", "Send me a taxi!" There are also another emerging applications [10].

The main goals of this paper are to provide the description of the TM based on assumption that it is an independent mobile terminal application, which can integrate different distributed external e-services by managing appropriate transactional processes. For that we use the ontology-based framework for transaction management so that the TM will be able to manage transaction across multiple e-services. We also consider concerns of atomicity protection of transactions within such TM architecture.

## 2 Ontology-based Transaction Monitor

Here we provide some background and the implementation basic for a TM based on ontologies.

### 2.1 Concept of Ontology-Based Transaction Management

The implementation of the TM we are basing on the assumption that the highest level of control functions related to transactions will be in hands of a user i.e. they should be implemented at the mobile terminal. This means that a user will be able to decide when to begin new transaction, when to stop or cancel it, when to switch from one service to another during the transaction, which values of parameters to use when submitting queries to a service and so on. Thus TM itself will be placed in a mobile terminal.

In general case we suppose that a user probably will need to contact several e-services to perform one transaction. Service better than user knows its recent offerings and the order of actions, which a user should do to get the service, he needs. This means that all interactions within one service will be better managed by a service itself. Such set of interactions we will call as a subtransaction and left the monitoring of it to a service. However we are leaving to a user the right to cancel active subtransaction and, due to atomicity requirements, return to the state, which was before the subtransaction started.

Mobile terminal should be able to manage situations when the contact with one service inside the transaction might be necessary to get an information, which is required as an input to deal with another service within the same transaction. To handle such cases a user should be sure that the Ids of such cross-parameters are the same for different services. For example, if one service returns to a user as output parameter "terminal location" and after that a user switches to LBS asking for a map around his location, then the LBS should recognize the input "terminal location" by the same way as the first service does.

To make possible to the TM to handle multiple service transactions and standardize e-services for that, we are presenting the concept of *ontology-based transaction management* (Figure 1) for implementation of the TM.

Every client in Figure 1, which in our case is mobile terminal, is equipped with a TM. Monitor was registered to several services and keeps basic *service data* about them, e.g. brief description, Id, contact address, the recent state of the monthly bill for the use of appropriate service and so on. Client also keeps data about active transaction, e.g. current state of parameters, active subtransaction, last query and so on.

Every service in Figure 1 is equipped with a Subtransaction Monitor, which allows to a service to work with multiple clients and know current state of a subtransaction with each of them. This Monitor manages stored at the service basic data about clients, e.g. logins, passwords, contact addresses, the recent state of an active subtransaction with this client, recent value of monthly bill of this client and so on. Monitoring is based on a *service tree*, which keeps an order of basic service actions, offered by a service to its clients, and appropriate states of possible subtransactions with this service.

The core of the approach is ontologies (Figure 1). Ontologies should be placed both in mobile terminals and in services, which is actually our case, or should be easily accessed by both from a third party. Ontologies define common multiple clients - multiple services standards and vocabularies for the use of the names, types, schemas, default values for parameters, atomic service actions with appropriate structure of their input and output. In our implementation ontologies help to the TM to deal with multiple services during transactions and simplify appropriate user interface.

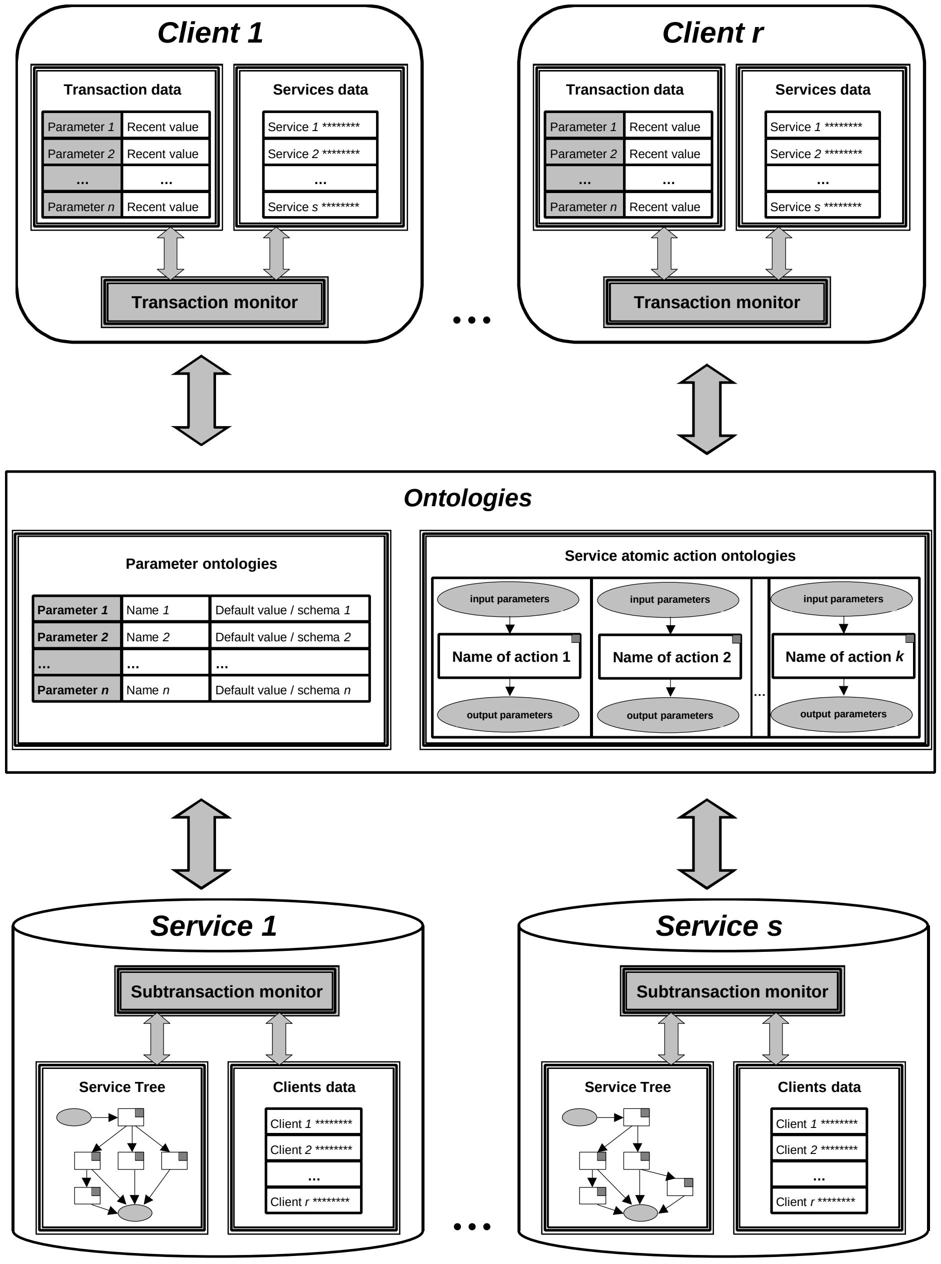


**Figure 1.** The conceptual scheme of the ontology-based transaction management

## 2.2 Some basic definitions

Let an *action* be a single client-server query-response session between the mobile terminal (hereinafter - terminal) and the e-service provider (hereinafter - service) as following:

$$A_i(x_1, x_2, ..., x_p, x_{p+1}, x_{p+2}, ..., x_{p+q}),$$

where $A_i$ is action's Id; $x_1, x_1, ..., x_p$ - Ids of *p* input parameters for the action, which should be specified at the terminal to create a query; $x_{p+1}, x_{p+2}, ..., x_{p+q}$ - Ids of *q* output parameters of the action, which the terminal receives as the result to its query.

*Subtransaction* $STR_i$ is a vector of one or more actions between a terminal and the service $Serv_j$ and appropriate states $S_0, ..., S_k$ managed by the service with definitely stated final goal and common memory of parameters:

$$STR_i : \{S_0; LOGIN[S_1], A_1[S_2], A_2[S_3], ..., A_{k-1}[S_k], LOGOUT[S_0]\}_{Serv_j},$$

where $S_0 = 0$ is an initial state of the subtransaction, $A_i[S_{i+1}]$ means that after performing the action $A_i$ the subtransaction will come to state $S_{i+1}$, *LOGIN* and *LOGOUT* are two obligatory actions, which are marginal for every service.

*Transaction* $TR_l$ is a vector of one or more subtransactions with the same terminal $Term_f$ and possibly different services managed by the terminal, with definitely stated final goal and common memory of parameters:

$$TR_l : \{STR_1, STR_2, ..., STR_r\}_{Term_f}.$$

*Service tree* is a tree-like structure of the set of subtransactions, which a service can offer to his clients and which is used by a service to manage subtransactions with clients (Figure 2). *Action of interest*, toned for every subtransaction in the service tree in Figure 2, is such an action, which outcome is in particular interest of a customer and has an economic value.

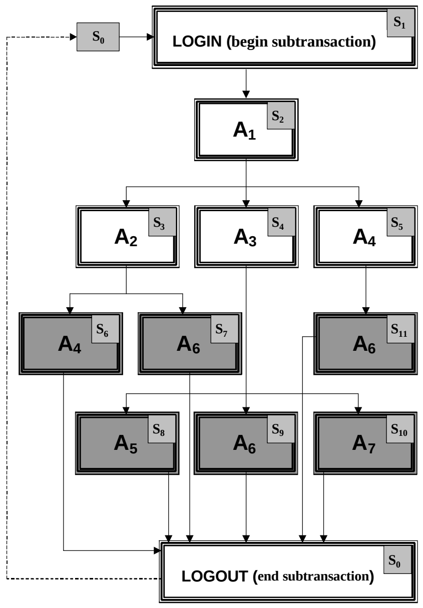


**Figure 2.** An example of a Service tree as a collection of subtransactions offered by the Service to its customers. In the rectangles together with the Id of an action there is also Id of a state, into which an appropriate subtransaction is coming after performing this action

## 2.3 Constants, ontologies and variables

In the TM model we consider a group of constants, which are defined by initial settings of the monitor (See Table 1). The group consists of:

1) *basic constants*, which define Ids of the terminal and services used, basic screens for the interface, total numbers of services, actions and parameters, which TM is operating with;
2) *service atomic actions ontologies* define basic actions with their input and output, from which every service can be composed, and which are used as a common procedural language between a client and a service (include always *LOGIN* and *LOGOUT* actions ontologies);
3) *parameter ontologies* describe parameters, which can be used in actions, by providing their Ids, default values and types (or schemas), and which are actually a common declarative language between a client and a service.

**Table 1.** Basic constants and ontologies of the TM

| ID of the Constant | Dimension | Value |
|---|---|---|
| ***Basic constants:*** | | |
| TERMINAL_ID | 1 | From settings |
| TOTAL_NUMBER_OF_SERVICES | 1 | From settings |
| TOTAL_NUMBER_OF_ACTIONS | 1 | From settings |
| TOTAL_NUMBER_OF_PARAMETERS | 1 | From settings |
| SERVICE_ID | TOTAL_NUMBER_OF_SERVICES | From settings |
| SCREEN_FRAME | 16 | From settings |
| ***Service atomic action ontologies:*** | | |
| ACTION_ID | TOTAL_NUMBER_OF_ACTIONS | From settings |
| INPUT_PARAMETERS_FOR_ACTION | TOTAL_NUMBER_OF_ACTIONS × TOTAL_NUMBER_OF_PARAMETERS | From settings |
| OUTPUT_PARAMETERS_FROM_ACTION | TOTAL_NUMBER_OF_ACTIONS × TOTAL_NUMBER_OF_PARAMETERS | From settings |
| ***Parameter ontologies:*** | | |
| PARAMETER_ID | TOTAL_NUMBER_OF_PARAMETERS | From settings |
| PARAMETER_DEFAULT_VALUE | TOTAL_NUMBER_OF_PARAMETERS | From settings |
| PARAMETER_TYPE/SCHEMA | TOTAL_NUMBER_OF_PARAMETERS | From settings |

In the TM model we consider three groups of variables:

1) *control variables* (Table 2) have sense only for a TM and are used to manage different states of the terminal during going-on transactions, subtransactions and actions;

2) *working variables* (Table 3) are used to manage parameters' states and provide common memory for different subtransactions, which can be run with different services. PARAMETER_CANCEL_ SUBTRANSACTION_VALUE is used to guarantee atomicity of a subtransaction (if for some reason a subtransaction cannot normally be finished, then the value of each parameter from the very beginning of the subtransaction will be restored);
3) *billing variables* (Table 4) are used to manage billing data in the TM. The terminal will collect bills separately for every service adding online price for appropriate service actions to it, when it is requested.

**Table 2.** Control variables of the TM

| ID of the Control Variable | Dimension | Initial Value |
|---|---|---|
| CURRENT_STATE_OF_TRANSACTION | 1 | 0 |
| CURRENT_STATE_OF_SUBTRANSACTION | 1 | 0 |
| LIST_OF_AVAILABLE_ACTIONS | TOTAL_NUMBER_OF_ACTIONS | 0 |
| ACTIVE_ACTION_ID | 1 | 0 |
| ACTIVE_PARAMETER_ID | 1 | 0 |
| ATOMICITY_PROTECTOR | 1 | 0 |

**Table 3.** Working variables of the TM

| ID of the Working Variable | Dimension | Initial Value |
|---|---|---|
| PARAMETER_RECENT_VALUE | TOTAL_NUMBER_OF_ PARAMETERS | PARAMETER_DEFAULT_VALUE |
| PARAMETER_CANCEL_SUBTRANSACTION _VALUE | TOTAL_NUMBER_OF_ PARAMETERS | PARAMETER_DEFAULT_VALUE |
| SCREEN | 1 | Screen 1 |

**Table 4.** Billing variables of the TM

| ID of the Billing Variable | Dimension | Initial Value |
|---|---|---|
| BILL_RECENT_VALUE | TOTAL_NUMBER_OF_SERVICES | 0 |
| PRICE_FOR_LAST_ACTION | 1 | 0 |

### 2.4 Actions: query-response sessions

*Service action* in our model is a single query-response session. Formats of service queries, which a mobile terminal can submit to a service and appropriate responses are given in Figure 3 (a-b). Being a part of a subtransaction this query-response session change a subtransaction state from one to another, according to a service tree. There are also *control actions* possible to be used to protect the subtransaction atomicity. Appropriate query-response formats are presented in Figure 3 (c-f).

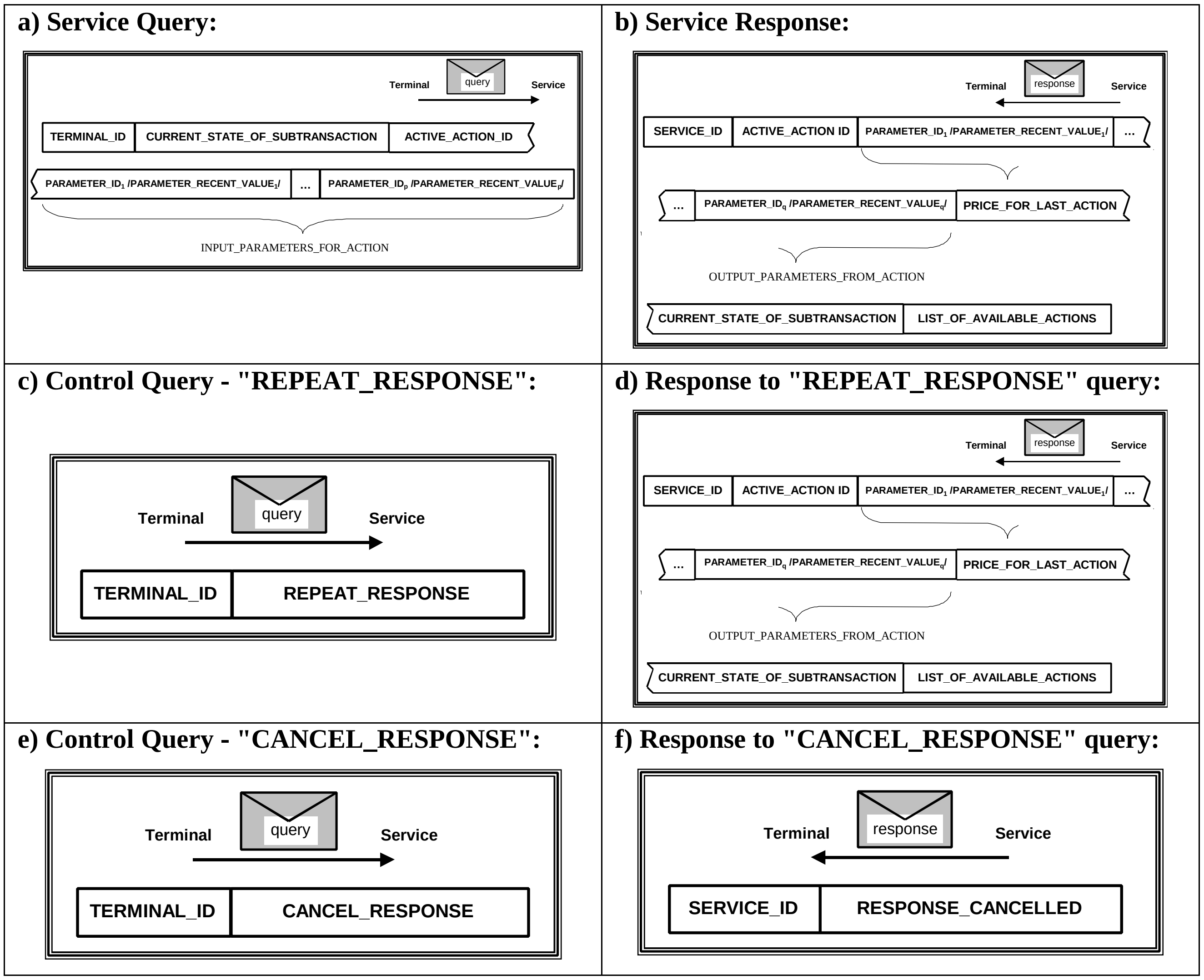

a) Service Query:
Terminal
query
Service
TERMINAL_ID
CURRENT_STATE_OF_SUBTRANSACTION
ACTIVE_ACTION_ID
PARAMETER_ID1 /PARAMETER_RECENT_VALUE1/
...
PARAMETER_IDp /PARAMETER_RECENT_VALUEp/
INPUT_PARAMETERS_FOR_ACTION
b) Service Response:
Terminal
response
Service
SERVICE_ID
ACTIVE_ACTION ID
PARAMETER_ID1 /PARAMETER_RECENT_VALUE1/
...
PARAMETER_IDq /PARAMETER_RECENT_VALUEq/
PRICE_FOR_LAST_ACTION
OUTPUT_PARAMETERS_FROM_ACTION
CURRENT_STATE_OF_SUBTRANSACTION
LIST_OF_AVAILABLE_ACTIONS
c) Control Query - "REPEAT_RESPONSE":
Terminal
query
Service
TERMINAL_ID
REPEAT_RESPONSE
d) Response to "REPEAT_RESPONSE" query:
Terminal
response
Service
SERVICE_ID
ACTIVE_ACTION ID
PARAMETER_ID1 /PARAMETER_RECENT_VALUE1/
...
PARAMETER_IDq /PARAMETER_RECENT_VALUEq/
PRICE_FOR_LAST_ACTION
OUTPUT_PARAMETERS_FROM_ACTION
CURRENT_STATE_OF_SUBTRANSACTION
LIST_OF_AVAILABLE_ACTIONS
e) Control Query - "CANCEL_RESPONSE":
Terminal
query
Service
TERMINAL_ID
CANCEL_RESPONSE
f) Response to "CANCEL_RESPONSE" query:
Terminal
response
Service
SERVICE_ID
RESPONSE_CANCELLED

**Figure 3.** Formats for query (a) and response (b) of a service action in terms of constants, ontologies and variables. Formats for query (c) and response (d) for a control action "REPEAT_RESPONSE" and the same e-f for a control action "CANCEL_RESPONSE".

An example of two service actions performed is shown in Figure 4.

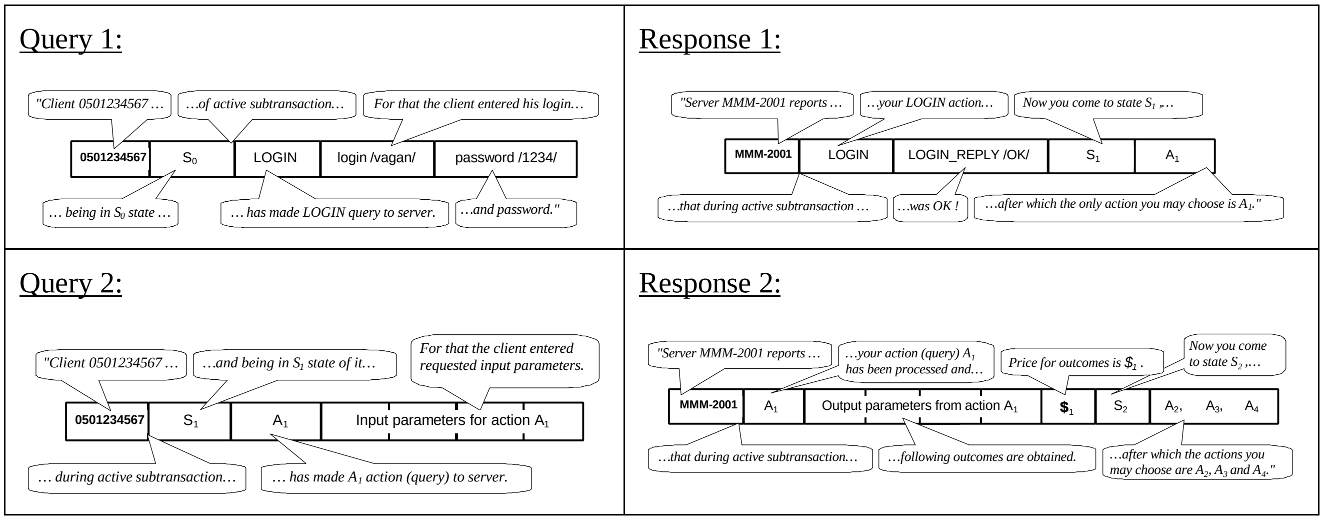
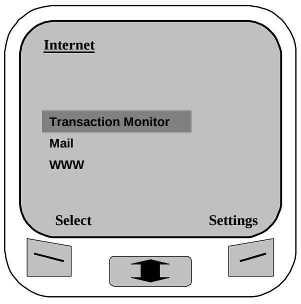

Query 1:
"Client 0501234567 …
…of active subtransaction…
For that the client entered his login…
0501234567
S0
LOGIN
login /vagan/
password /1234/
… being in S0 state …
… has made LOGIN query to server.
…and password."
Response 1:
"Server MMM-2001 reports …
…your LOGIN action…
Now you come to state S1 ,…
MMM-2001
LOGIN
LOGIN_REPLY /OK/
S1
A1
…that during active subtransaction …
…was OK !
…after which the only action you may choose is A1."
Query 2:
"Client 0501234567 …
…and being in S1 state of it…
For that the client entered requested input parameters.
0501234567
S1
A1
Input parameters for action A1
… during active subtransaction…
… has made A1 action (query) to server.
Response 2:
"Server MMM-2001 reports …
…your action (query) A1 has been processed and…
Price for outcomes is $1 .
Now you come to state S2 ,…
MMM-2001
A1
Output parameters from action A1
$1
S2
A2, A3, A4
…that during active subtransaction…
…following outcomes are obtained.
…after which the actions you may choose are A2, A3 and A4."

**Figure 4.** An example of two performed actions (client-server query-response sessions) between the Terminal and the Service. These are first two actions of the subtransaction according to the Service Tree from Figure 2

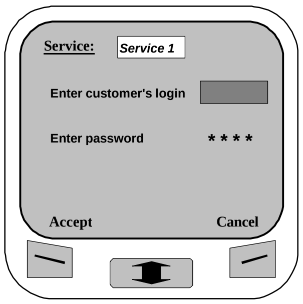
## 2.5 A user interface

Figure 5 presents terminal screens from the implementation of the TM.

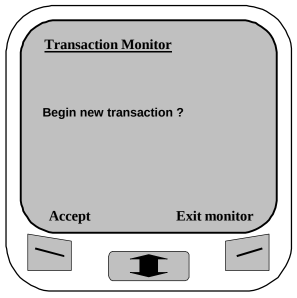
Screen 1: Selecting Transaction Monitor



Screen 2: Logging in to the service

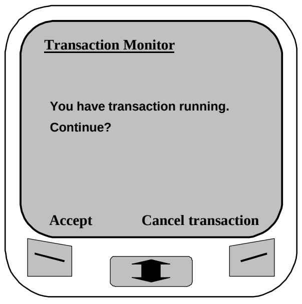


Screen 3: Starting new transaction



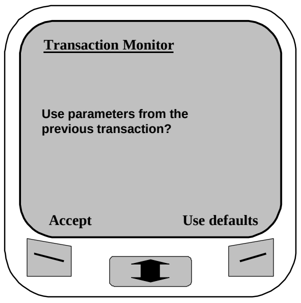
Begin new transaction ?

Screen 4: Continuing active transaction



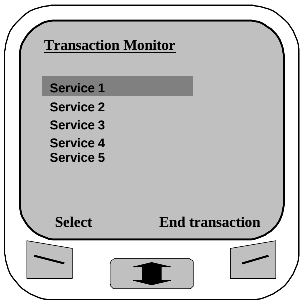
You have transaction running. Continue?

Screen 5: Deciding which parameters to use for a new transaction

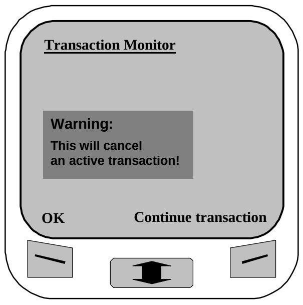


Use parameters from the previous transaction?

Screen 6: Selecting a service

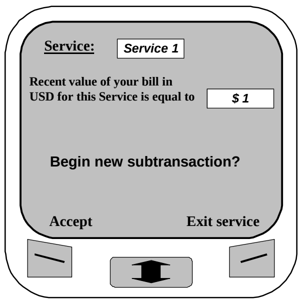


Screen 7: Warning concerning transaction cancellation



Warning:
This will cancel an active transaction!

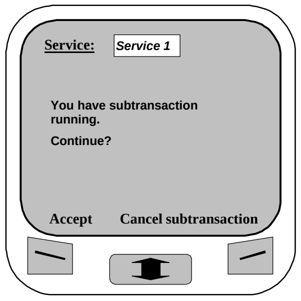
Screen 8: Deciding to begin a new subtransaction



Recent value of your bill in USD for this Service is equal to $ 1

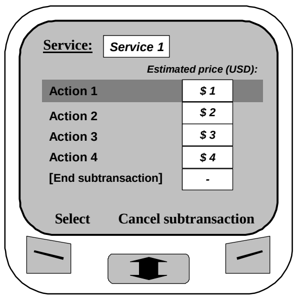
Begin new subtransaction?

Screen 9: Deciding to continue an active subtransaction



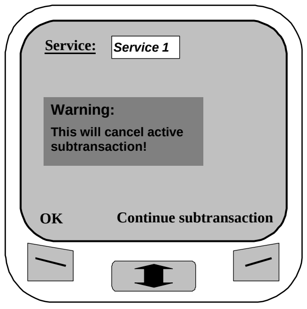
You have subtransaction running.
Continue?

Screen 10: Selecting an action within the service



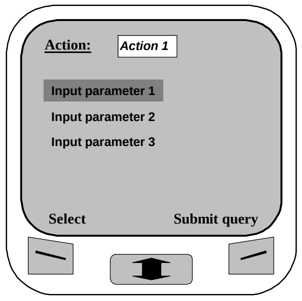
| | Estimated price (USD): |
|---|---|
| Action 1 | $ 1 |
| Action 2 | $ 2 |
| Action 3 | $ 3 |
| Action 4 | $ 4 |
| [End subtransaction] | - |

Screen 11: Warning about subtransaction cancellation

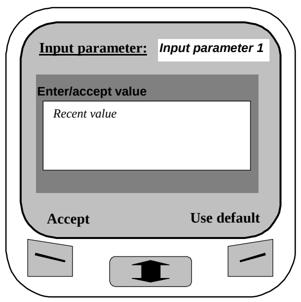


Warning:
This will cancel active subtransaction!

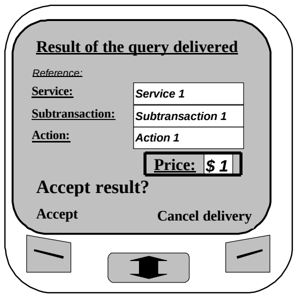
Screen 12: Selecting input parameter of an action for viewing/ editing



Screen 13: Editing an input parameter



Screen 14: Accepting the result and price of action delivered from the service

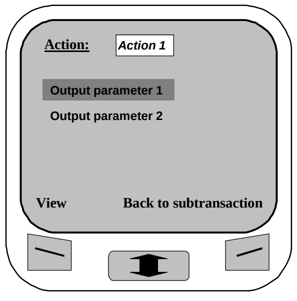


| | |
|---|---|
| Service: | Service 1 |
| Subtransaction: | Subtransaction 1 |
| Action: | Action 1 |

Price: $ 1

Accept result?

Screen 15: Selecting the delivered output parameter for viewing/ editing



Screen 16: Viewing/ editing the delivered output parameter

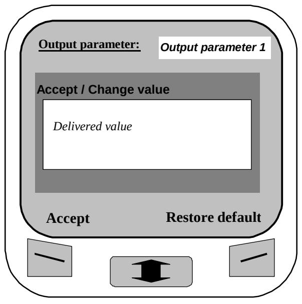


**Figure 5.** Terminal screens from the implementation of the TM

In Figure 6 the scheme of algorithm for the TM is presented. Nodes in that scheme are states of Monitor presented by the appropriate terminal screens. Links show the transitions from state to state depending on user's selection of appropriate control buttons at the terminal.

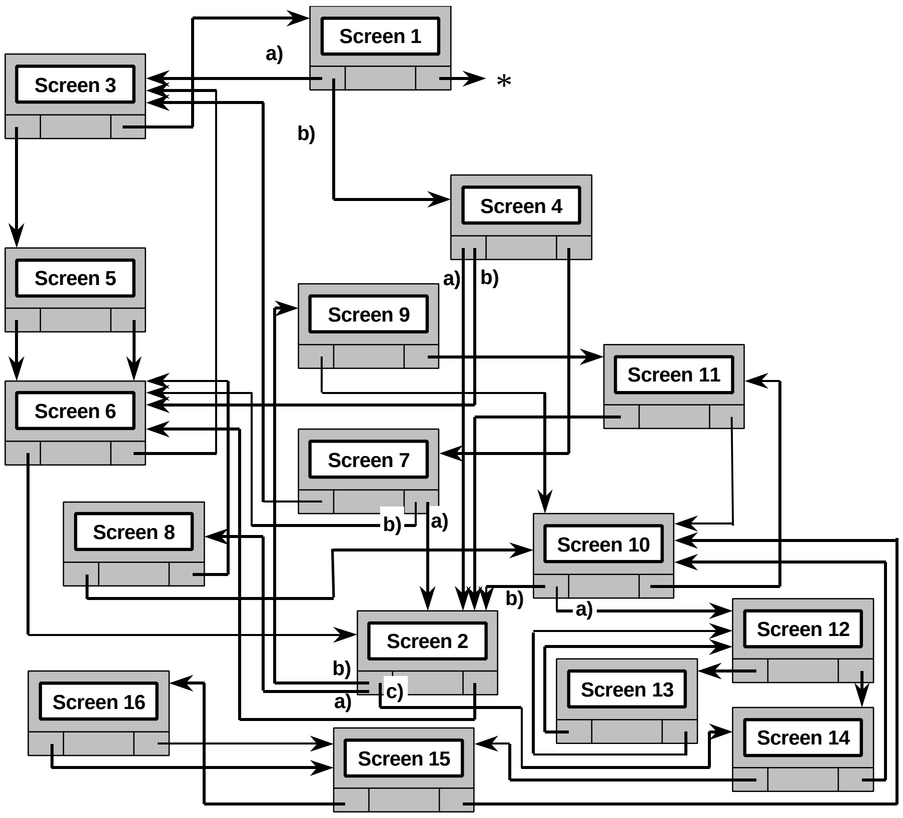


**Figure 6.** The scheme of algorithm for the TM with basic terminal screens and transitions

## 3 Transaction atomicity protection

Atomicity is a transactional property that describes the existence relation between tasks, which is either all tasks in the group should execute or no task in the group should execute. In other words an atomic unit of work is either executed completely or not at all [2].

A transaction should be executed in full or not at all. If executed in full, the transaction is committed. A transaction may however also be aborted due to input errors, system overloads, deadlocks, or system crashes. Atomicity requires that, if a transaction is aborted, its partial results should be undone (roll-back). The activity of preserving the transaction's atomicity in presence of explicit aborts is called transaction recovery. The activities of ensuring atomicity in the event of system crashes are called crash recovery [11].

In [12] the atomicity issues were expanded from e- to m-commerce context and include:

- *Money atomicity:* Money is either entirely transfer or not transfer at all;
- *Goods atomicity:* Customer receives the ordered goods if and only if merchant is paid;
- *Distributed Purchase Atomicity:* Products bought from different suppliers are either both delivered or none.

Atomicity of an action is protected by special parameter ATOMICITY_PROTECTOR, which initially assigned to 0. When a terminal sends a service query to the service it also immediately changes this parameter to 1 and then the terminal will keep this value until receives appropriate response from the service. The break of a subtransaction in the middle of an action, i.e. between a query and a response will be recognized and handled by sending REPEAT_RESPONSE query to the service.

Atomicity of subtransaction, i.e. a session with one separate service, is protected by *action of interest* framework and *default delivery confirmation* method.

*Action of interest* within a subtransaction is an only action, which outcome is in particular interest of a customer and has an economic value. Service tree of every service should be organized in such a way that only such action in every possible subtransaction requests payment from a customer, which summarizes all expenses of this service to complete all other actions in this subtransaction. Action of interest guarantees atomicity of a subtransaction, i.e. if a customer accepts just this action and is being billed for it, then he is certainly has in hands what he is expected from the whole subtransaction.

*Default delivery confirmation* method used in this model means that when goods are arrived and a customer knows about this he is already billed for that by default. Only if a customer is not canceling the delivery, then he can use the goods. Only if the delivery is cancelled then the service will be automatically informed about this and will make roll-back of the customers bill. In that cancellation case the Transaction Monitor will not physically allow to the customer to use delivered goods.

More complicated issue is an atomicity of the transaction as whole in the context of multiple services because it also adds *distributed purchase atomicity* requirement.

In many E-Commerce applications, interaction of customers is not limited to a single merchant. Consider, for instance an example in Figure 7, where a customer wants to purchase specialised software (SW) from a merchant. In order run this software, he also needs an operating system (OS), which is, however, only available from a different merchant. As both goods individually are of no value for the customer, he needs the guarantee to perform the purchase transaction with the two different merchants atomically in order to get either both products or none.

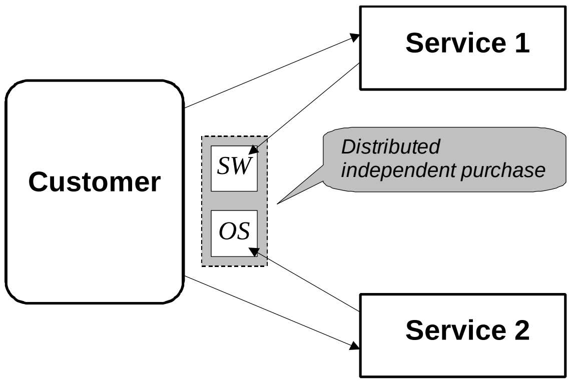


**Figure 7.** Distributed Purchase case where both parts of the target good can be purchased independently

*Distributed purchase atomicity* addresses the encompassment of interactions with different independent merchants into one single transaction. Most currently deployed payment co-ordinators support only money atomicity while some advanced systems address also distributed purchase atomicity. However, all three dimensions are – to our best knowledge – not provided by existing systems and protocols although the highest level of guarantees would be supported and although this is required by a set of real-world applications. This lack of support for full atomicity in E-Commerce payment is addressed in [8] where authors apply transactional process management to realise an E-Commerce Payment Co-ordinator.

At the example in Figure 7 we have case when both purchases were "physically" independent on each other. Theoretically customer can make them in different order or simultaneously. To fully guarantee distributed purchase atomicity in such case Transaction Monitor should be located at some external "Middleman", which for example was used in [11] as Transaction Service, or as above [8] in the form of E-Commerce Payment Co-ordinator.

However in our model we are dealing with even more complicated business cases when purchases are "physically" dependent on each other in a way that one cant start new purchase without fully completing previous one, i.e. outcome of that first purchase is one of necessary requirements to start the second one. Their appearance in the same transaction has sense if we assume that the outcome of the first purchase itself has no value for a customer - only as necessary step to get target good from the second purchase. Consider example in Figure 8.

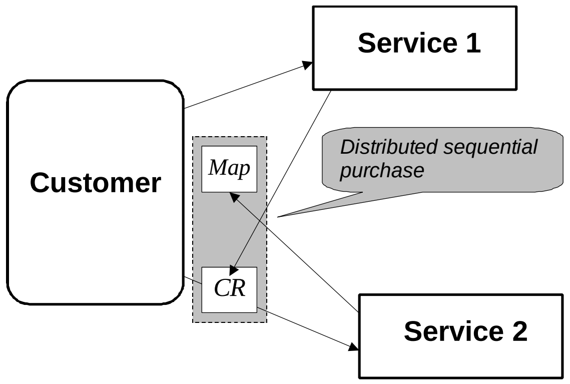


**Figure 8.** Distributed Purchase, where goods from distributed services are purchased in a certain sequence

Assume that a customer needs a Map from Service 2 but to apply for that map he is requested to provide his coordinates (CR). Coordinates he can get from Service 1. Assume that Service 1 does not care about how a customer is going to use coordinates delivered - the service has made job and got money for it. Even if the rest of a transaction will fail and for some reason a customer will not get his Map from Service 2, full compensation for the transaction as whole cannot be guaranteed. However even in this case for atomicity protection can be used the *roll-back technique*.

Some transaction mechanisms use roll-back techniques to restore some previous state and the re-play the actions subsequent to reaching that state [11]. Roll-back is simply a particular kind of recovery strategy of the broader concept of “compensation” in which compensatory actions are used to reverse the effects of failed actions by exploiting semantic knowledge. Compensation can be used when the actions have had real-world effects and there is no way to roll-back their effects.

## 4 Related work

According to MeT "Consistent User Experience" framework [3] the user interface should allow to people to transfer their knowledge and skills from one application to any other application. Consistency of visual interface and terminology helps people to learn and then easily recognize the "language" of the interface. The TM, due to implementation of the concept of ontology-based transaction management, offers such a consistent standardize user interface with multiple services.

E-speak is an open software platform designed specifically for the development, deployment, intelligent interaction, and management of globally distributed e-services [14]. E-Speak makes services capable to interact with each other on behalf of their users, and compose themselves into more complex services. The E-Speak Service Engine [14] actually is a TM software that performs the intelligent interaction of e-services. The TM in the hands of user is a good example of an E-Speak engine, which expands the E-Speak internet-based framework to wireless.

OntoWeb [6] - Ontology-based information exchange for knowledge management and electronic commerce is the IST Project Thematic Network of 64 academic and industrial partners. The goal of the OntoWeb Network is to bring together researchers and industrials promoting interdisciplinary work and strengthening the European influence on Semantic Web standardisation efforts based on RDF and XML. The implementation of the concept of ontology-based transaction management for mobile terminal allows expanding a target for the Semantic Web framework also to m-commerce.

## 5 Conclusion

In this paper we assume that TM is an independent mobile terminal application, which can integrate different distributed external e-services by managing appropriate transactional processes. For that we use the ontology-based framework for transaction management so that the TM will be able to manage transaction across multiple e-services securing atomicity if the transaction.

Another possible approach, which was described in [9] was based on the assumption that some specific terminal-based application is already exists, which supports certain transactions with certain services. In that case the TM was used as a tool to guarantee basic transactional properties of that transactions, i.e. protect the application's data from terminations related to the specifics of a mobile device. TM in this case will be fully controlled by the application.

With the increasing market of electronic commerce it becomes an interesting aspect to use autonomous mobile agents for electronic business transactions. Being involved in money transactions, supplementary security features for mobile agent systems have to be ensured. In [13] an architecture was presented for a mobile agent system which offers fault tolerance for the whole agent system at a high level. This architecture pretends to guarantee security for the host as well as security for the agent. To handle these issues for mobile agents we use various encryption mechanisms and a novel method was applied for mobile agent systems by using distributed transactions in the architecture. Due to this security architecture an agent will be enabled to carry out money transactions.

All three approaches seem to be reasonable to integrate to be used in the appropriate context.

## Acknowledgements

The economic support of the National Technology Agency of Finland (TEKES), Nokia, Hewlett Packard Finland, and Yomi Vision under contract 40599/99 is highly appreciated. Discussions and suggestions from Prof. J. Veijalainen and other colleagues are also highly appreciated.

## References

1. P. Bernstein, V. Hadzilacos, N. Goodman, Concurrency Control and Recovery in Database Systems, Addision-Wesley, 1987.
2. W. Derks, J. Dehnert, P. Grefen, W. Jonker, Customized Atomicity Specification for Transactional Workflows, TR-CTIT-00-24, University of Twente, The Netherlands, December 2000, 40 pp.
3. MeT Consistent User Experience, Version 1.0, 21 February 2001, available in: http://www. mobiletransaction.org.
4. MeT Overview White Paper, Version 2.0, 29 January 2001, available in: http://www.mobiletransaction.org /pdf/White%20Paper_2.0.pdf.
5. Mobile Electronic Transactions Forum, available in: http://www.mobiletransaction.org/.
6. OntoWeb: Ontology-Based Information Exchange for Knowledge Management and Electronic Commerce, 2000, available in: http://www.ontoweb.org.
7. C. Papadimitriou, The Theory of Database Concurrency Control, Computer Science Press, 1986.
8. H. Schuldt A. Popovici, H.-J. Schek, Give Me All I Pay for - Execution Guarantees in Electronic Commerce Payment Processes, In: Proc. of Informatik'99: Enterprise-Wide and Cross-Enterprise Workflow Management, Paderborn, Germany, October 1999, pp. 12-19.
9. V. Terziyan, J. Veijalainen, M-Commerce Transaction Model Implementation at a Mobile Terminal, Multimeetmobile Project Report, TITU, Univ. of Jyvaskyla, 9 May 2001, 56 pp.
10. V. Terziyan, J. Veijalainen, H. Tirri, Mobile e-Commerce Transaction Model, Multimeetmobile Project Report, TITU, Univ. of Jyvaskyla, 18 December 2000, 48 pp.
11. Transaction Services: Supporting Inter-Organisational Processes, GigaTS 1.1.3, Telematica Institute, Netherlands, November 1999, available in: http://www.telin.nl/dscgi/ds.py/Get/File-2876/D1.1.3v1_Transaction_Services.pdf.
12. J. Veijalainen, Transactions in Mobile Electronic Commerce, LNCS, Vol. 1773, Springer, 1999, pp. 208-229.
13. H. Vogler, T. Kunkelmann, M.-L. Moschgath, Distributed Transaction Processing as a Reliability Concept for Mobile Agents, 6th IEEE Workshop on Future Trends of Distributed Computing Systems, October 29-31, 1997.
14. What is E-Speak? Product Information, 2001, Hewlett Packard Company, available in: http://www.e-speak.hp. com/product/overview.shtm.