Multimeetmobile project

TIETOTEKNIIKAN
TUTKIMUSINSTITUUTTI

M-commerce transaction model implementation at a mobile terminal

31.03.2001

Vagan Terziyan

Jari Veijalainen

1 Introduction

The document is a deliverable of the Multimeetmobile (MMM) project performed by the Information Technology Research Institute, University of Jyväskylä, and financed by the Finnish National Technology Agency (TEKES), Hewlett Packard Finland, Nokia, and Yomi Vision. The main goals of this report are to provide the description of transactional mechanism (Transaction Monitor) for mobile business applications and its implementation plan. We consider two different approaches to the implementation of the Transaction Monitor (TM). First one is more general approach and it 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. For that we use the ontology-based framework for transaction management so that the Transaction Monitor 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 Transaction Monitor implementation. Another approach is based on the assumption that some specific terminal-based application is already exists, which supports certain transactions with certain services. In this case the TM is intended to be 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. Recent state of the MMM pilot system is just an example of such application, to support which we are implementing the application-driven TM. For both of approaches we presenting appropriate detailed software architectures for implementation.

2 Mobile e-commerce transactions

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.

M-commerce transactions are inherently distributed, because they are always performed over a wireless link and are thus protocol-driven. When we use the term in a technical sense, m-commerce transaction refers below to:

- specification of a m-commerce protocol and its overall semantics;
- execution of the protocol and the steps launched by the message exchanges at different players.

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

According to [9] one can distinguish the following m-commerce players: a user; mobile network operator (MNO); telecom operator; application provider; facility supplier; information provider; contents holder; solution provider; financial institution; terminal manufacturer.

In [11] we have considered five different types of m-commerce transactions: banking applications, Internet e-commerce transactions over a wireless access networks, location-based services, retail shopping over Bluetooth, and ticketing applications.

Location-based services (LBS) are more in the domain of the MNOs. The basic idea is 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 MNOs 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 [8].

3 Ontology-based Transaction Monitor and its implementation plan

Here we provide the implementation basic for a Transaction Monitor above the MMM pilot system.

3.1 Concept of Ontology-Based Transaction Management

The implementation of the Transaction Monitor 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 Transaction Monitor 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 Transaction Monitor 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 Transaction Monitor.

Every client in Figure 1, which in our case is mobile terminal, is equipped with a Transaction Monitor. 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 Transaction Monitor to deal with multiple services during transactions and simplify appropriate user interface.

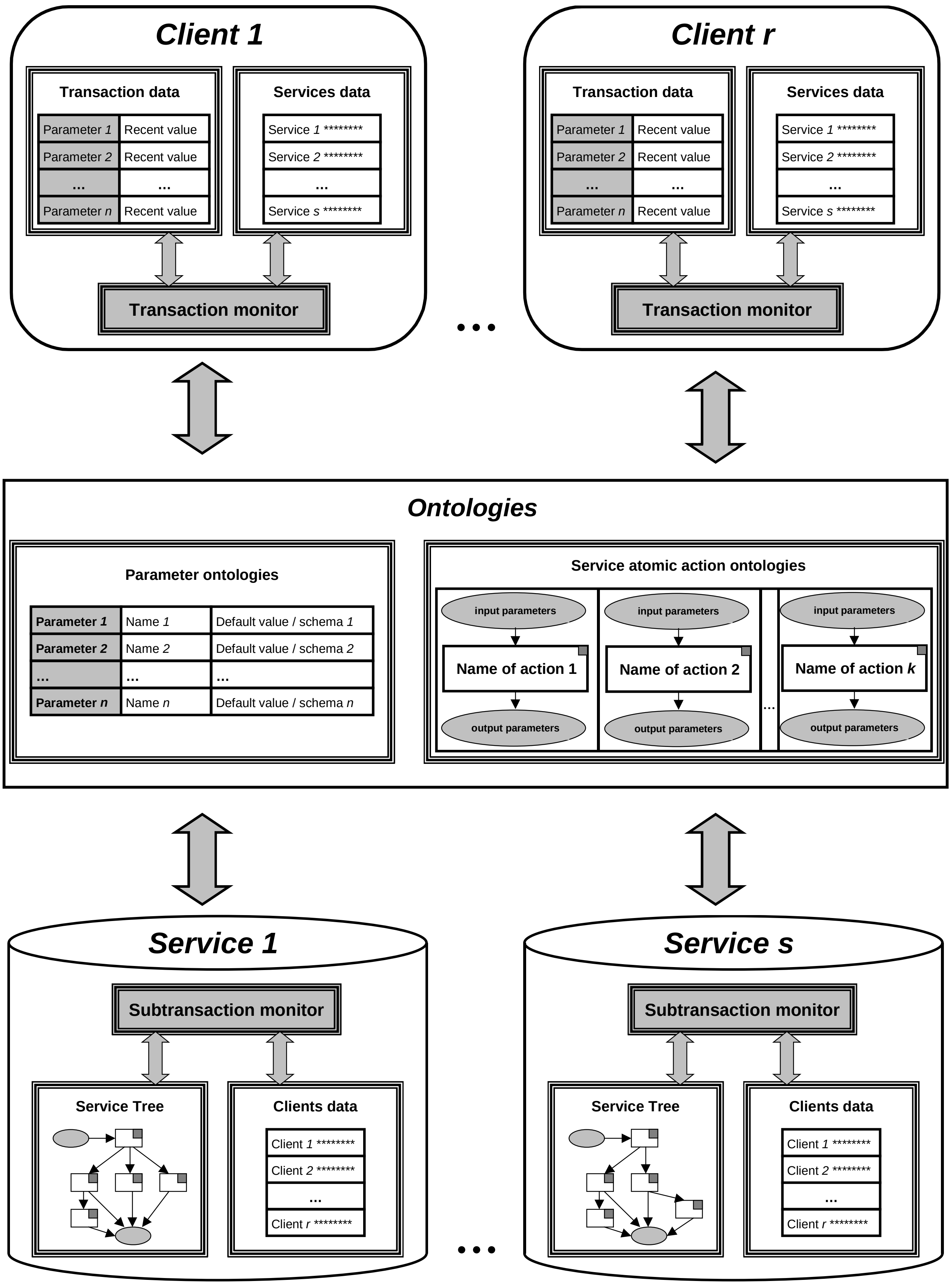


Figure 1. The conceptual scheme of the ontology-based transaction management

3.2 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.

3.3 Constants, ontologies and variables

In the Transaction Monitor 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 Transaction Monitor 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.

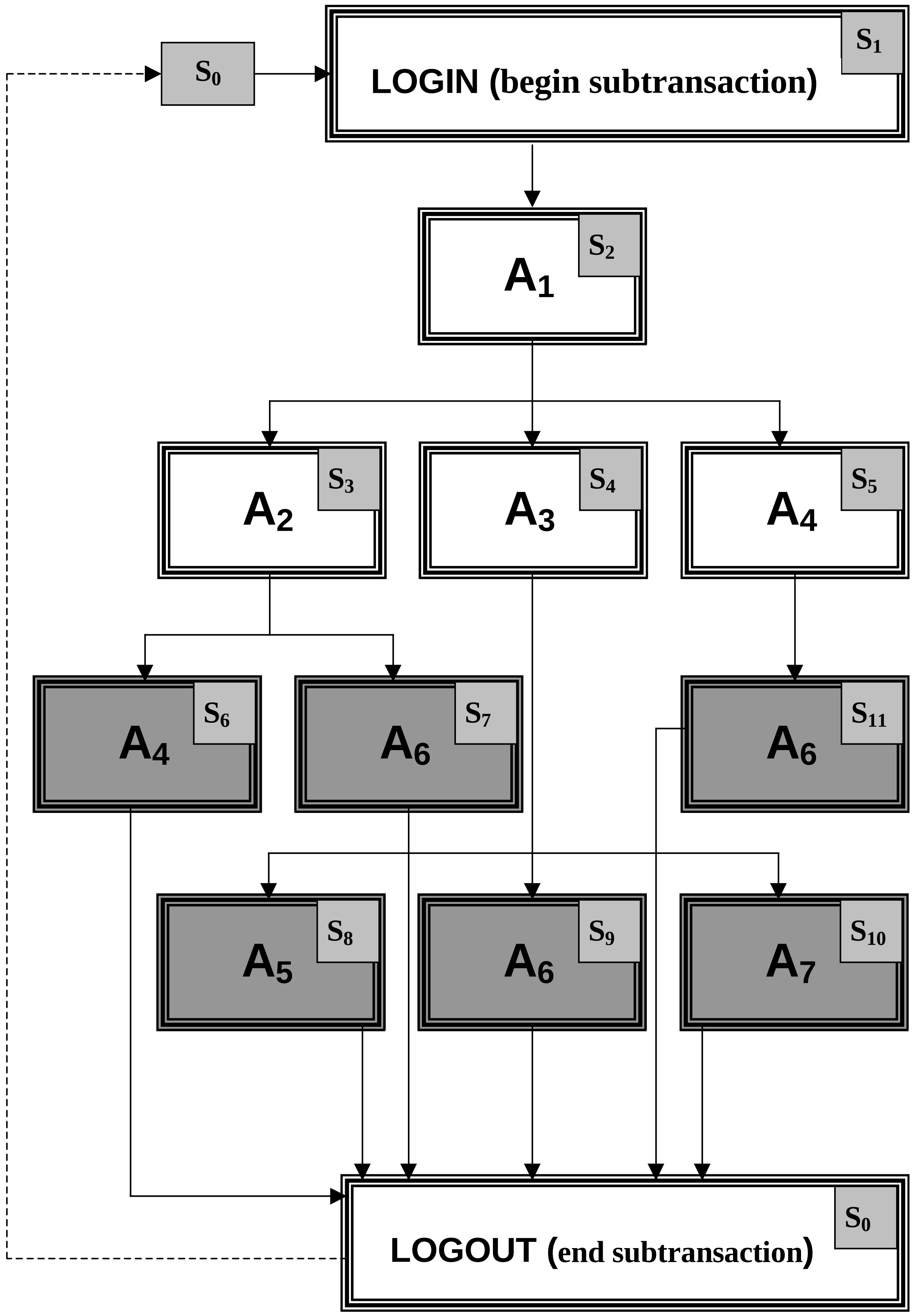


Figure 2. An example of a Service tree as a collection of subtransactions offered by the Service to its customers. In the rectangles together with the Id of an action there is also Id of a state, into which an appropriate subtransaction is coming after performing this action

Table 1. Basic constants and ontologies of the Transaction Monitor

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 Transaction Monitor model we consider three groups of variables:

1) *control variables* (Table 2) have sense only for a Transaction Monitor 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 Transaction Monitor. 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 Transaction Monitor

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 Transaction Monitor

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 Transaction Monitor

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

3.4 Actions: query-response sessions

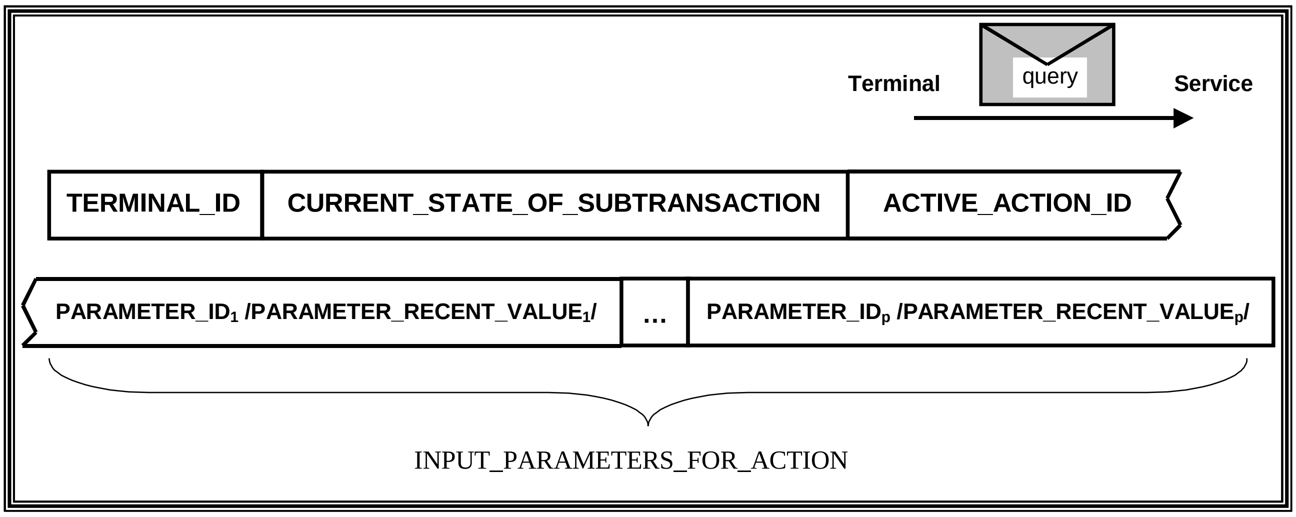
Service action in our model is a single query-response session. Formats of service queries, which a mobile terminal can submit to a service and appropriate responses are given in Figure 3 (a-b). Being a part of a subtransaction this query-response session change a subtransaction state from one to another, 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).

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

3.5 Terminal screens from a user interface

Figures 5 - 20 present terminal screens from the implementation of the Transaction Monitor.

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:

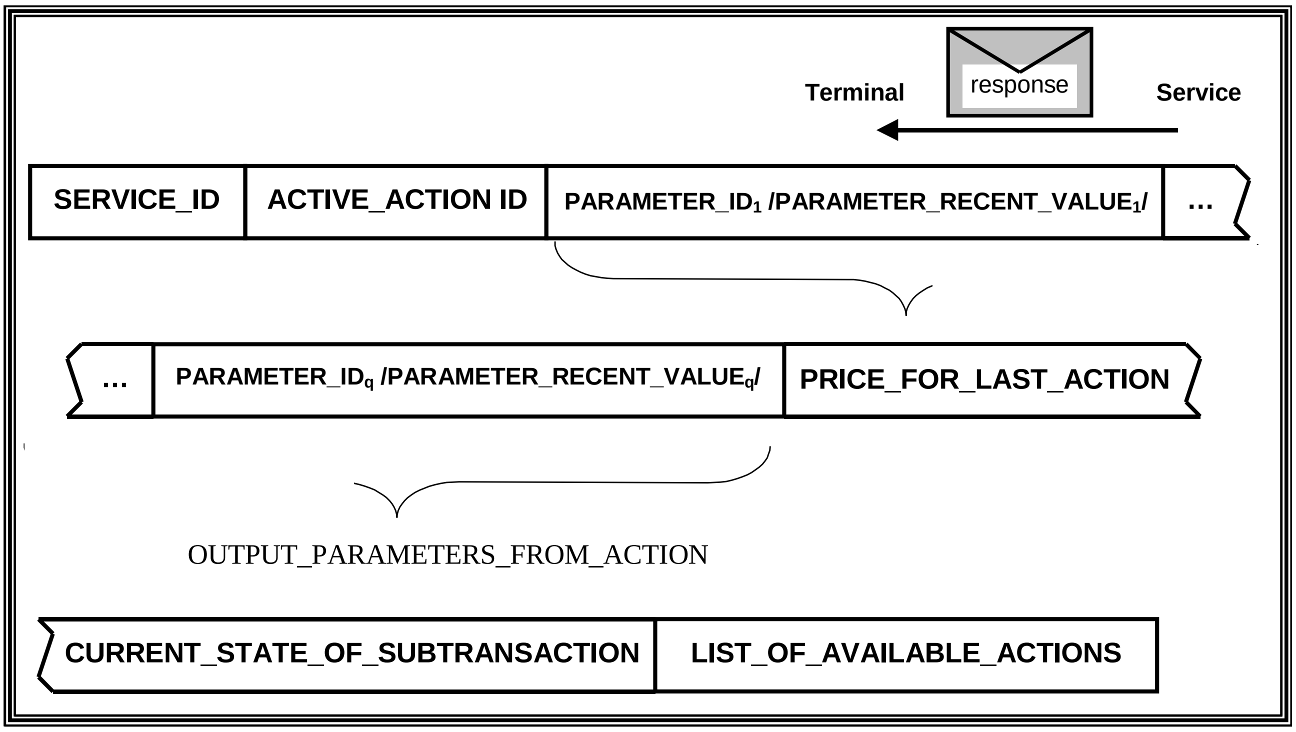
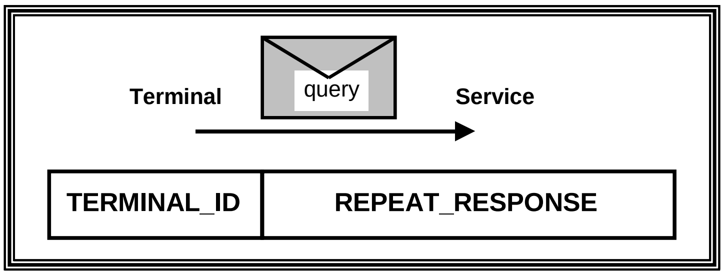
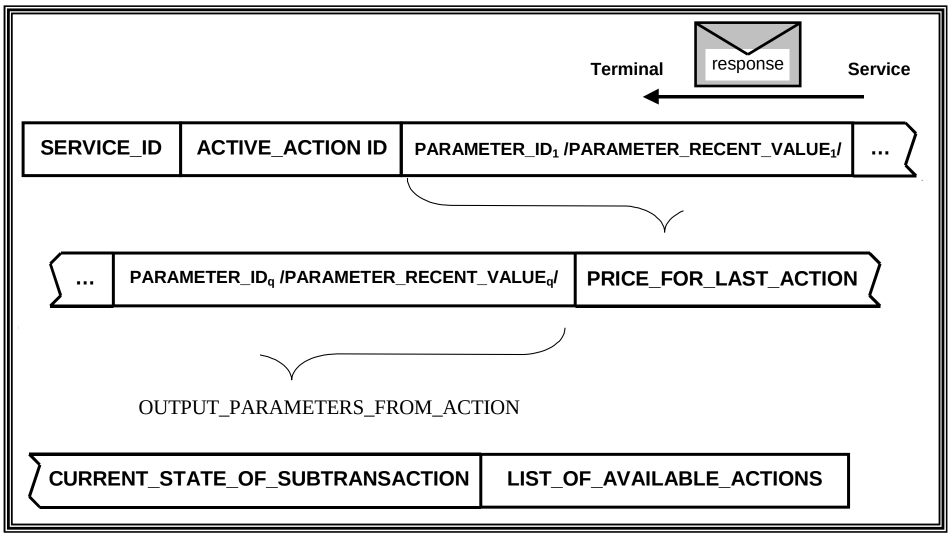

Terminal
response
Service
SERVICE_ID
ACTIVE_ACTION ID
PARAMETER_ID1 /PARAMETER_RECENT_VALUE1/
…
…
PARAMETER_IDq /PARAMETER_RECENT_VALUEq/
PRICE_FOR_LAST_ACTION
OUTPUT_PARAMETERS_FROM_ACTION
CURRENT_STATE_OF_SUBTRANSACTION
LIST_OF_AVAILABLE_ACTIONS

Figure 3. Formats for query (a) and response (b) of a service action in terms of constants, ontologies and variables

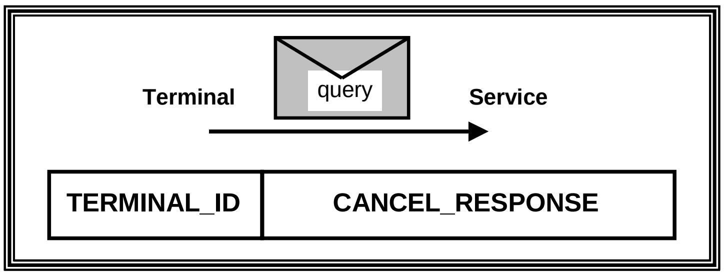
c) Control Query - "REPEAT_RESPONSE":



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":



f) Response to "CANCEL_RESPONSE" query:

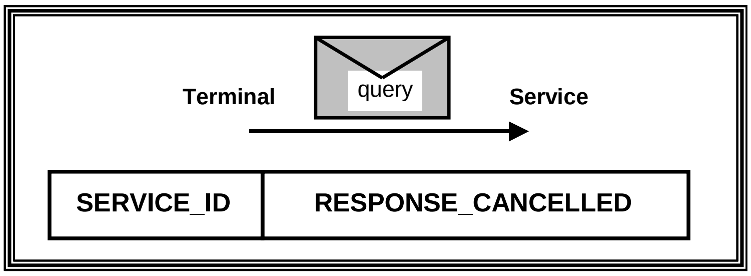


Figure 3 - continue. Formats for query (c) and response (d) for a control action "REPEAT_RESPONSE" and the same e-f for a control action "CANCEL_RESPONSE"

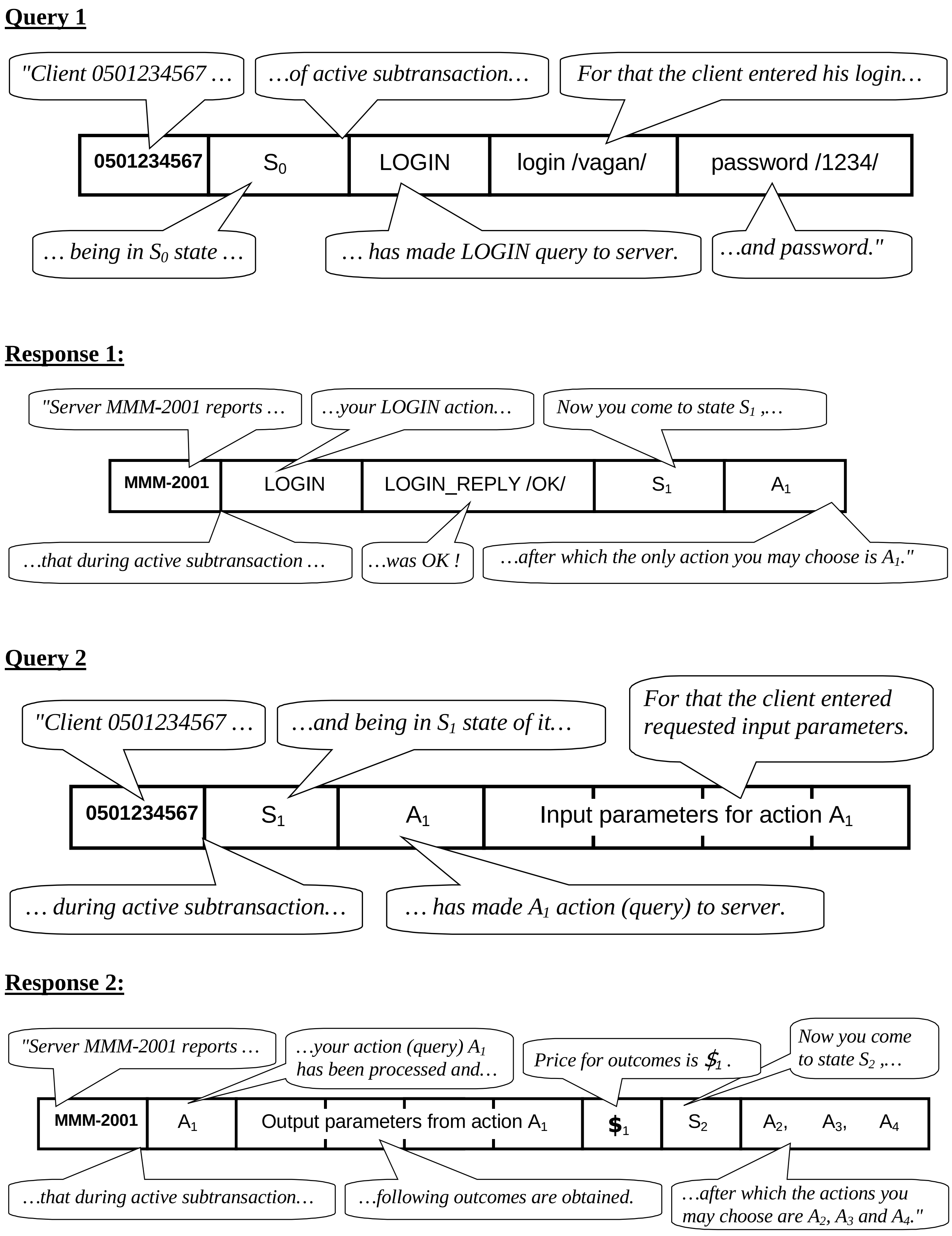

Query 1
"Client 0501234567 …
…of active subtransaction…
For that the client entered his login…
0501234567
S_0
LOGIN
login /vagan/
password /1234/
… being in S_0 state …
… has made LOGIN query to server.
…and password."
Response 1:
"Server MMM-2001 reports …
…your LOGIN action…
Now you come to state S_1 ,…
MMM-2001
LOGIN
LOGIN_REPLY /OK/
S_1
A_1
…that during active subtransaction …
…was OK !
…after which the only action you may choose is A_1."
Query 2
"Client 0501234567 …
…and being in S_1 state of it…
For that the client entered requested input parameters.
0501234567
S_1
A_1
Input parameters for action A_1
… during active subtransaction…
… has made A_1 action (query) to server.
Response 2:
"Server MMM-2001 reports …
…your action (query) A_1 has been processed and…
Price for outcomes is $\$_1$.
Now you come to state S_2 ,…
MMM-2001
A_1
Output parameters from action A_1
$\$_1$
S_2
A_2, A_3, A_4
…that during active subtransaction…
…following outcomes are obtained.
…after which the actions you may choose are A_2, A_3 and A_4."

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

Screen 1

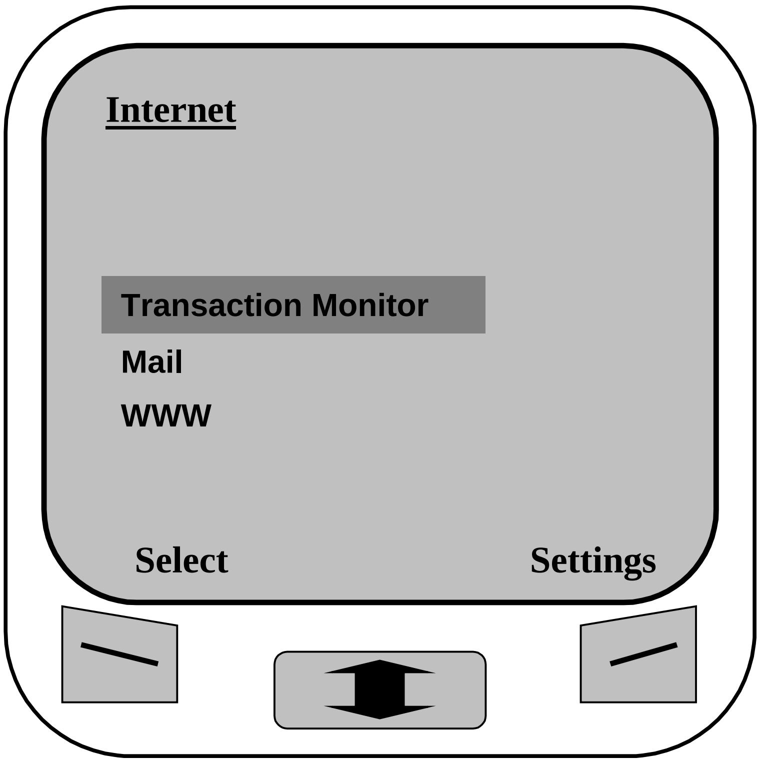


Figure 5. Selecting Transaction Monitor

Screen 2

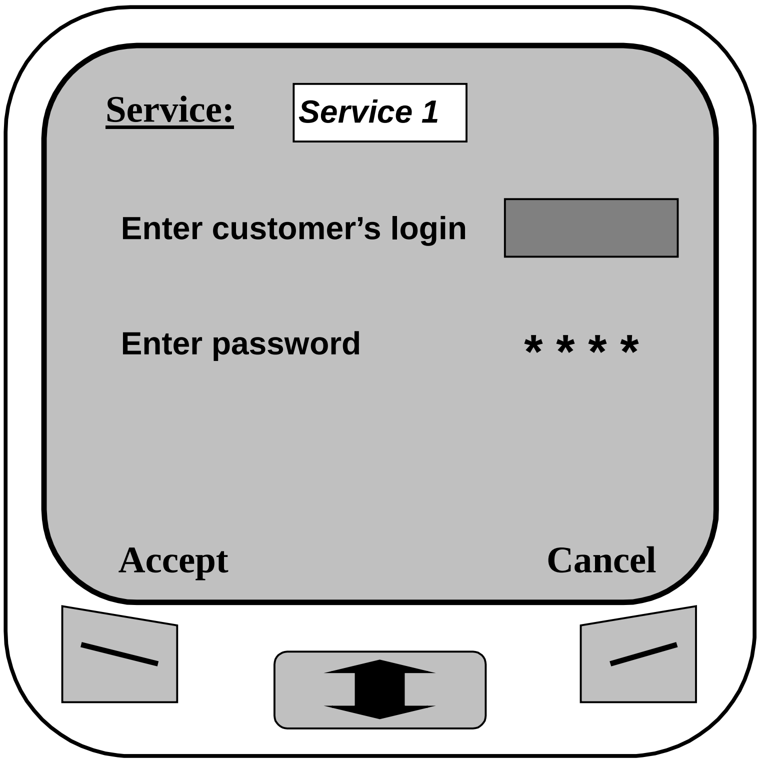


Figure 6. Logging in to the service

Screen 3

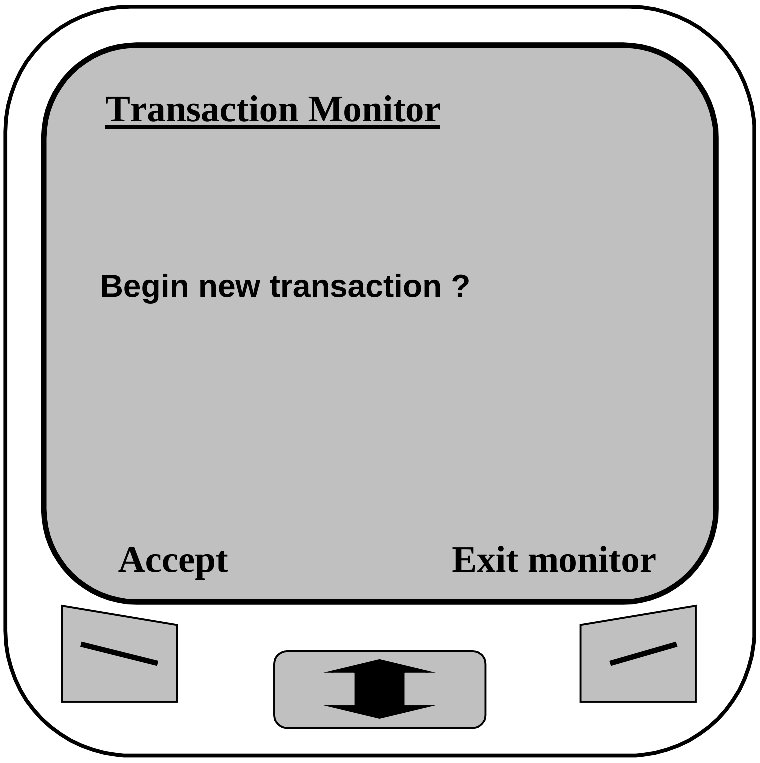


Figure 7. Starting new transaction with the Monitor

Screen 4

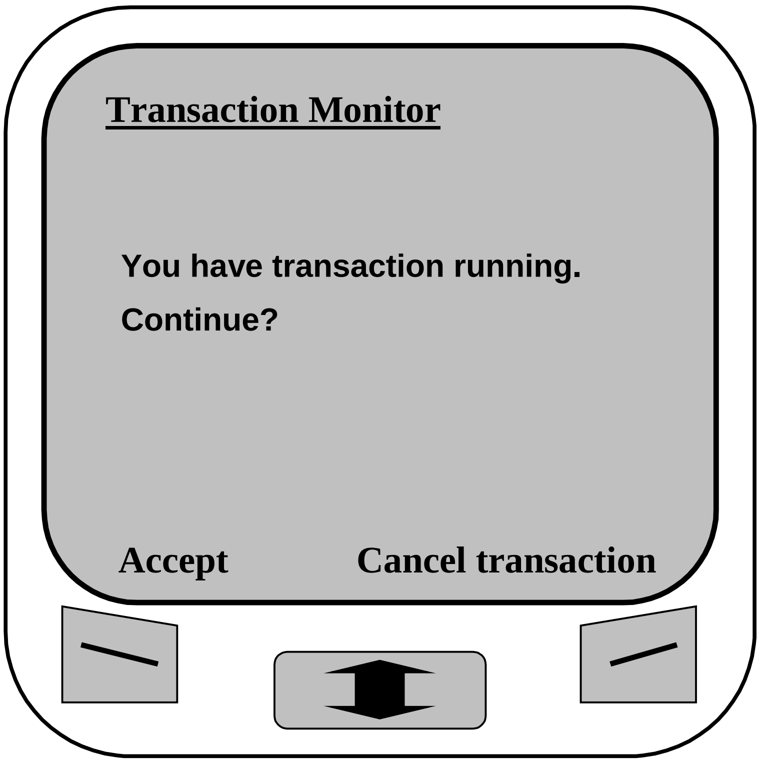
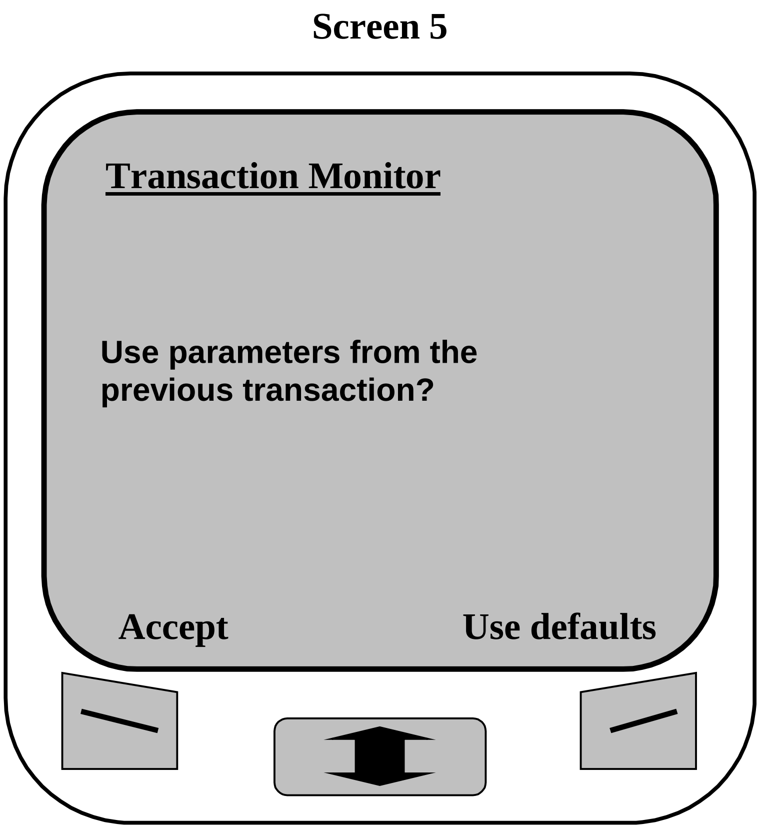


Figure 8. Continuing active transaction



Use parameters from the previous transaction?

Figure 9. Deciding the use of default or recent parameters for a new transaction

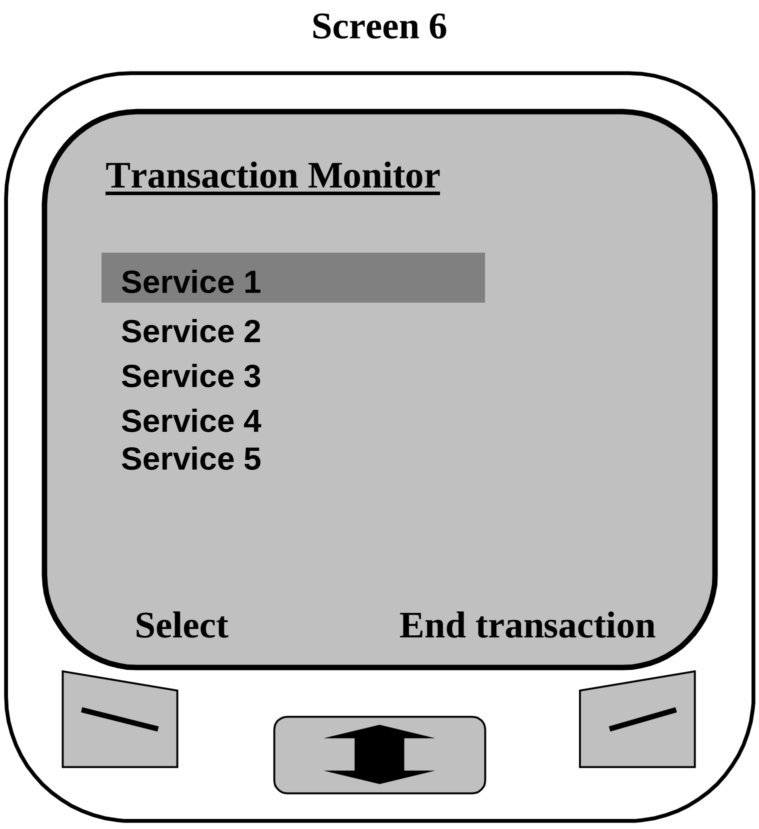


Figure 10. Selecting a service

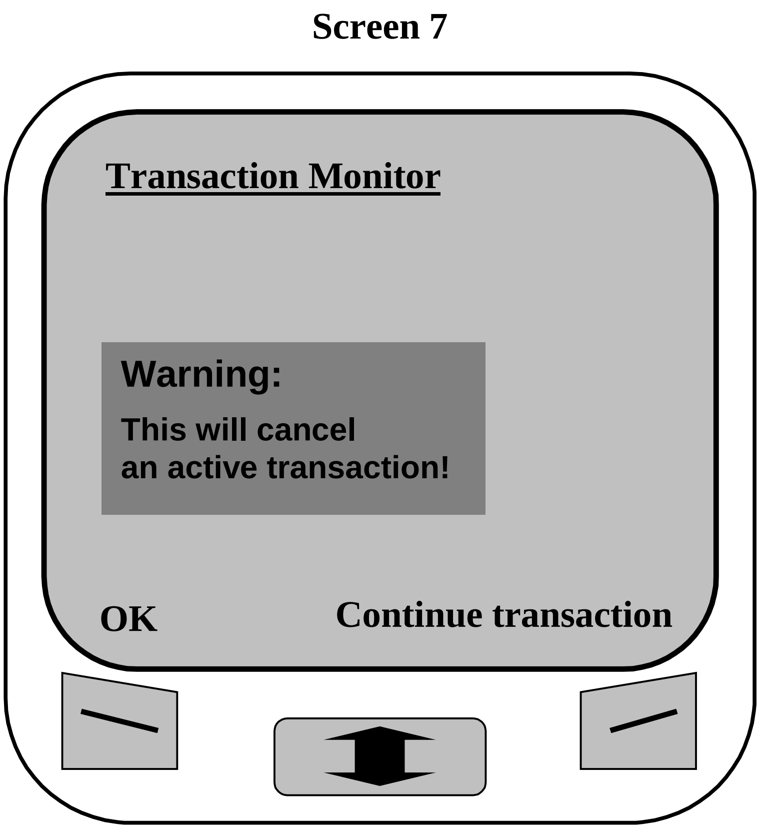


Figure 11. Warning concerning the transaction cancellation

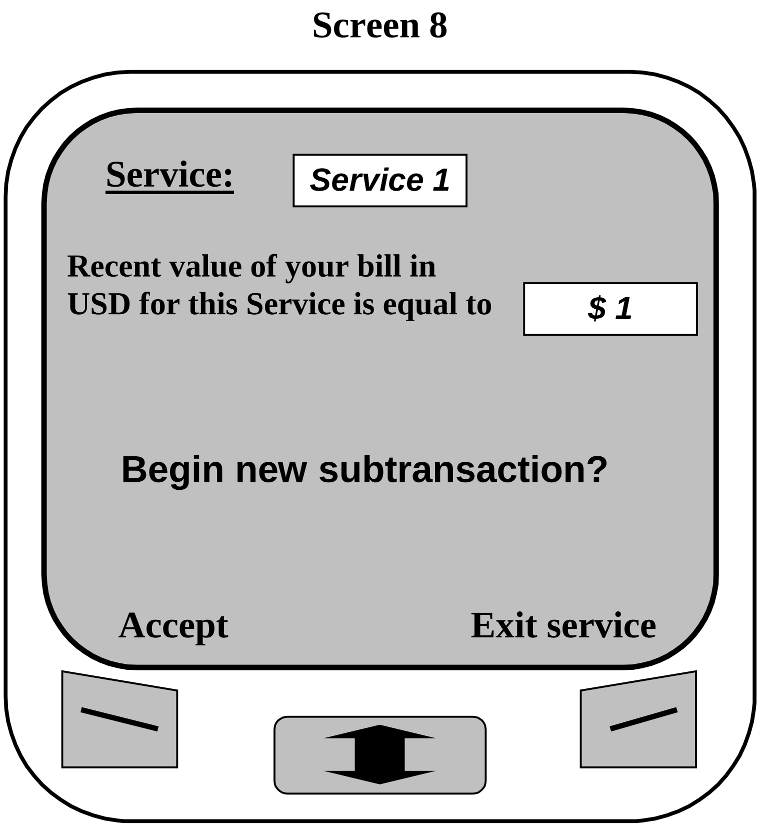

Screen 8
Service:
Service 1
Recent value of your bill in
USD for this Service is equal to
$ 1
Begin new subtransaction?
Accept
Exit service

Figure 12. Deciding to begin a new subtransaction

Screen 9

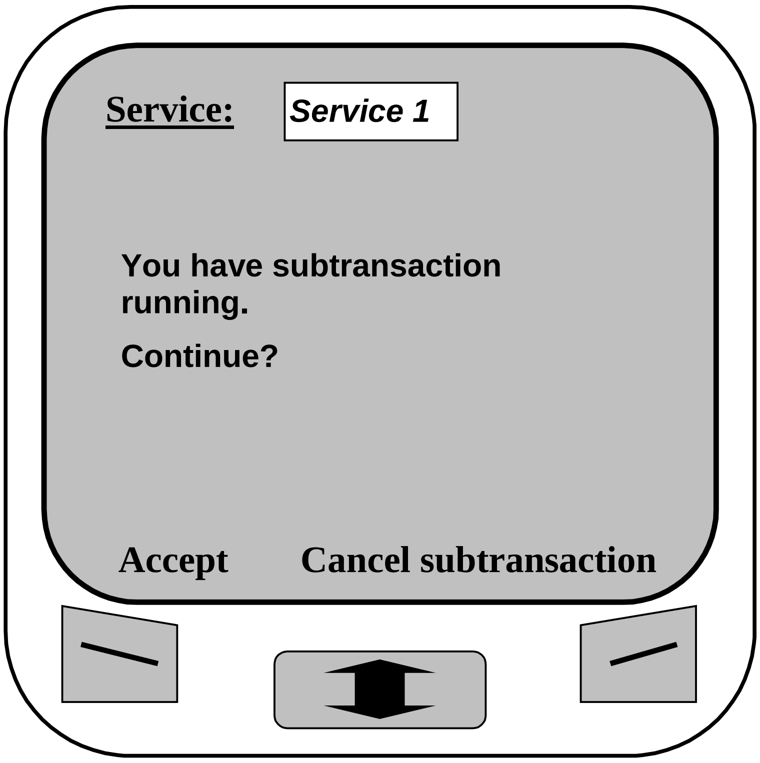


Figure 13. Deciding to continue an active subtransaction

Screen 10

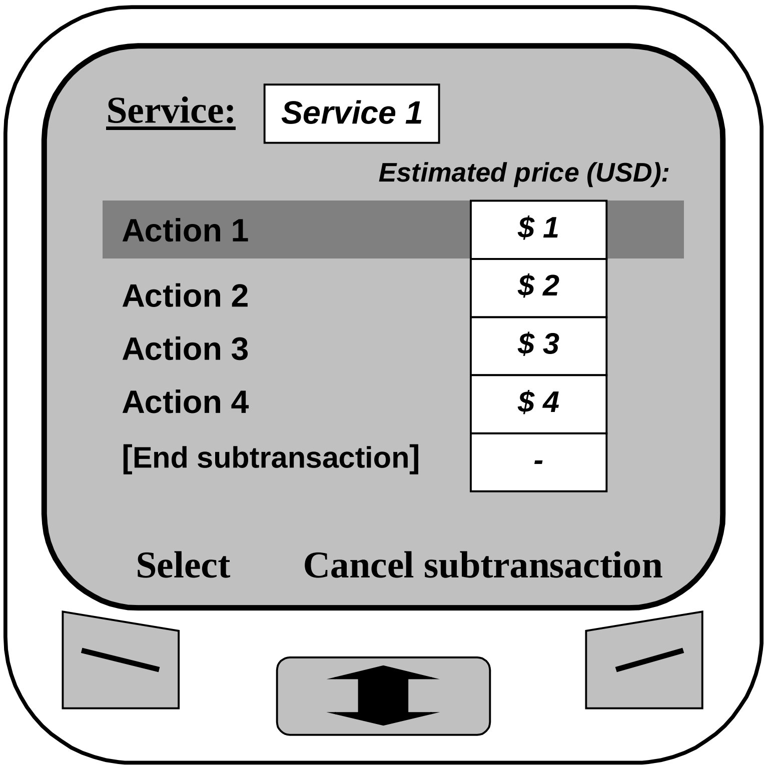


Figure 14. Selecting an action within the service

Screen 11

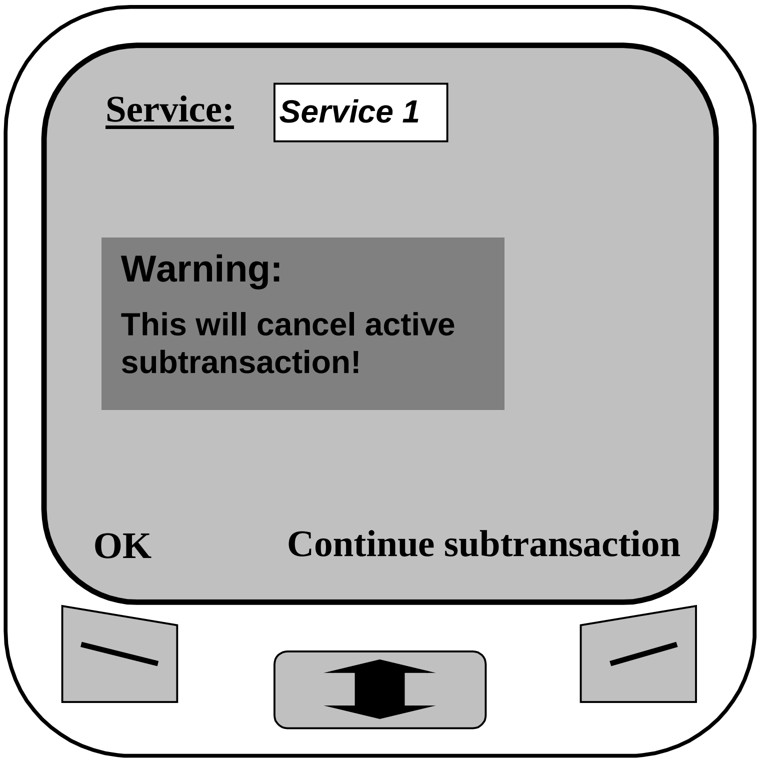


Figure 15. Warning concerning a subtransaction cancellation

Screen 12

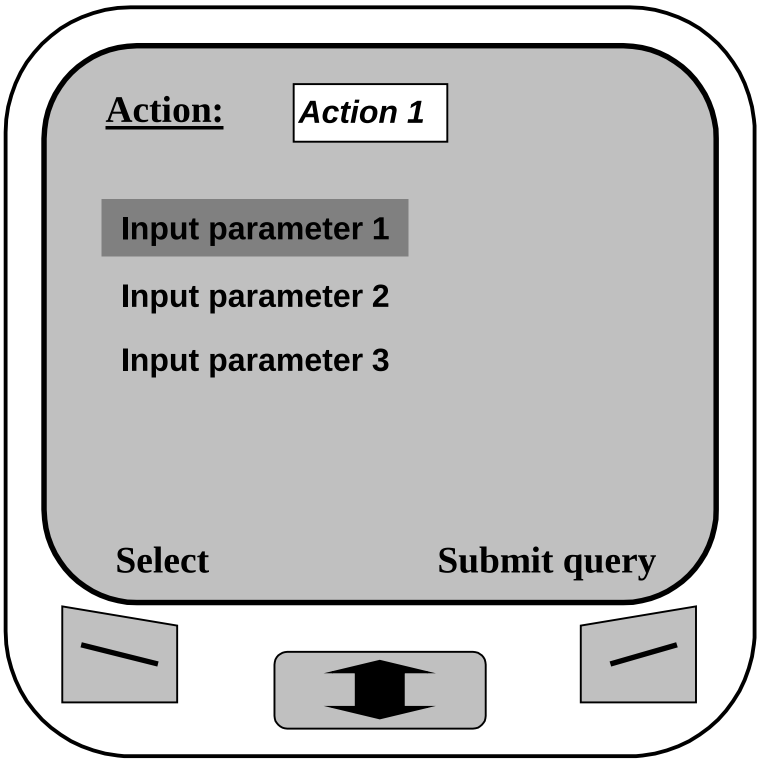


Figure 16. Selecting an input parameter for the action for viewing/editing

Screen 13

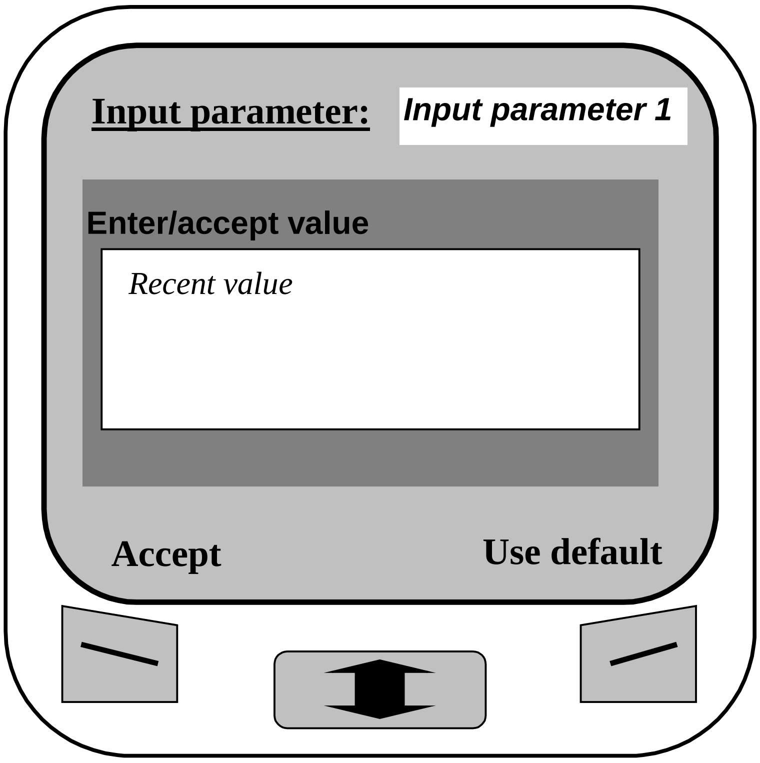


Figure 17. Editing an input parameter

Screen 14

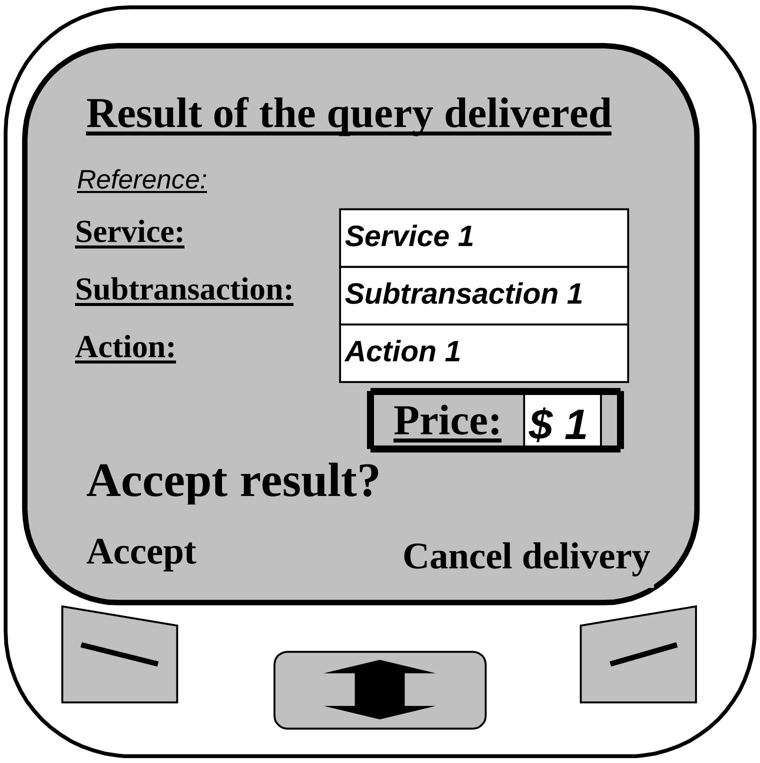


Figure 18. Accepting the result and price of action delivered from the service

Screen 15

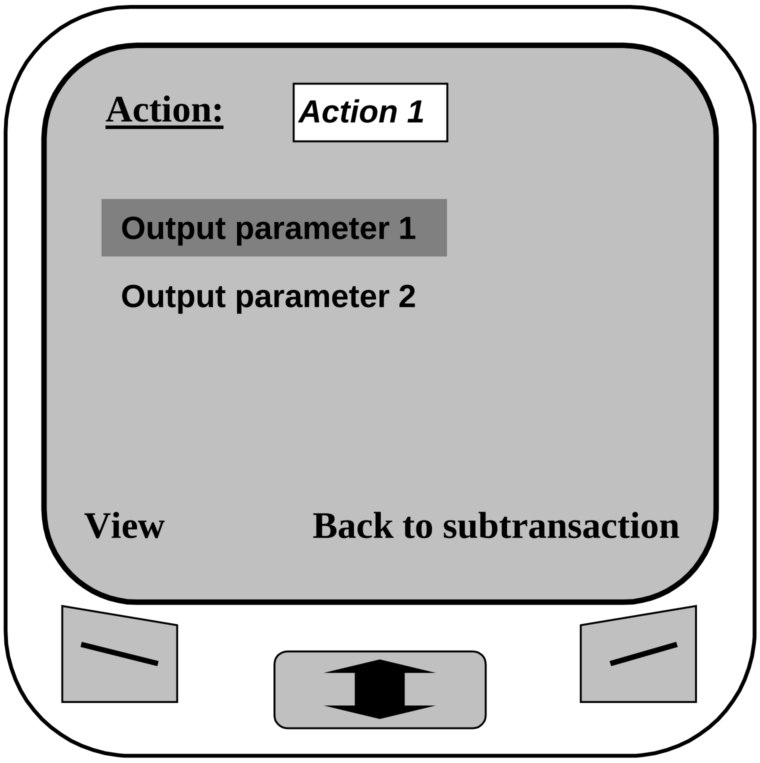


Figure 19. Selecting the delivered output parameter for viewing/editing

Screen 16

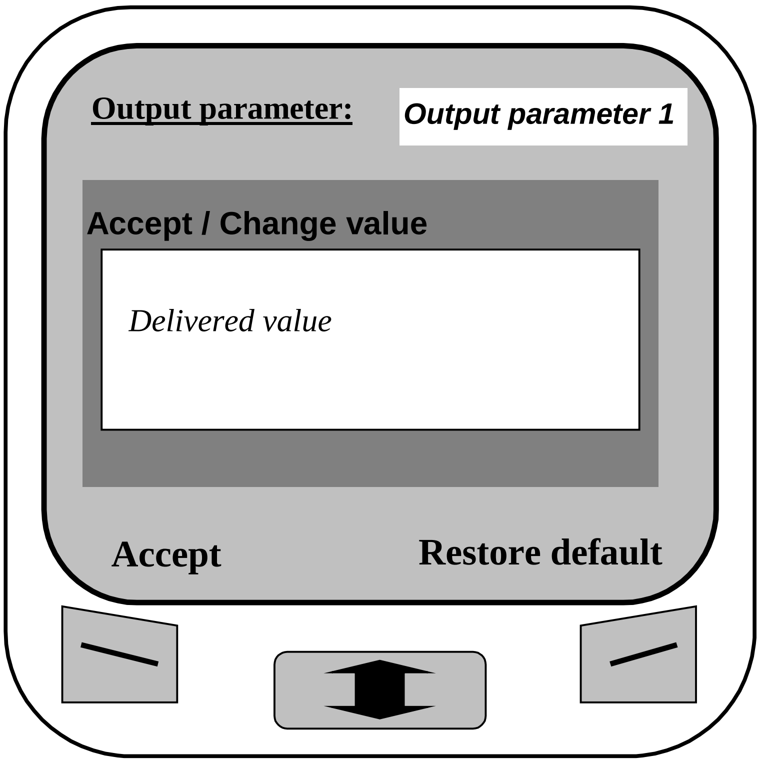


Figure 20. Viewing/editing the delivered output parameter

3.6 Algorithm for the Transaction Monitor

In Figure 21 the scheme of algorithm for the Transaction Monitor 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.

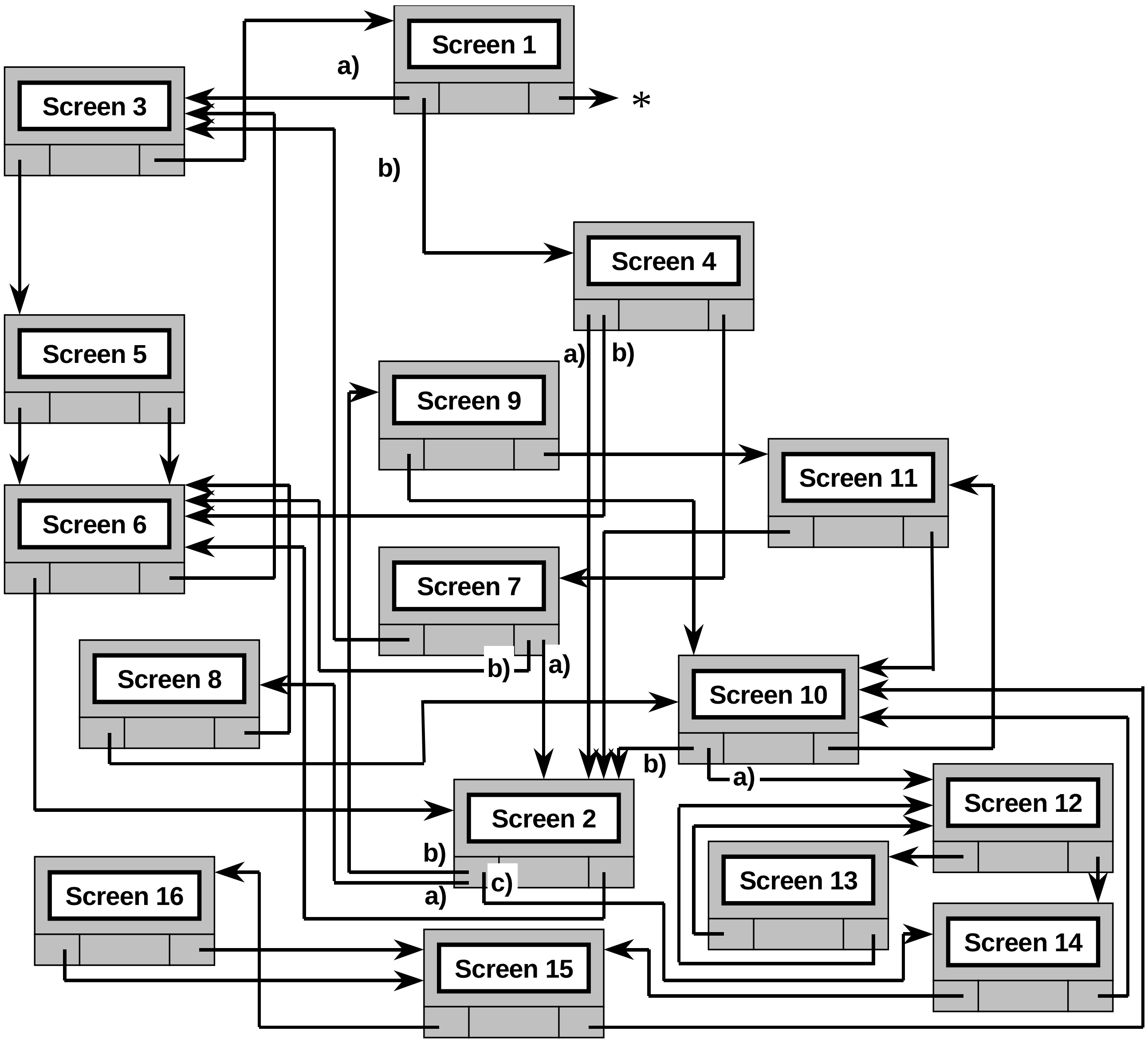


Figure 21. The scheme of algorithm for the Transaction Monitor with basic terminal screens and transitions

The following Figures 22 - 58 shows each of the transitions separately and the description of each of them with description of associated parameters changes is given in the following text.

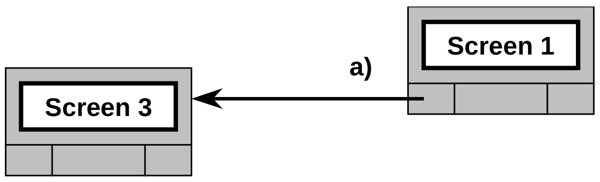


Figure 22. Screen 1 - Screen 3 -transition (a)

When a user selects the **Transaction Monitor** from Screen 1 (Figure 5) and terminal knows that there is no transaction still running, i.e. CURRENT_STATE_OF_TRANSACTION = 0, then terminal displays Screen 3 (Figure 7) and offers to a user to start new transaction (Figure 22).

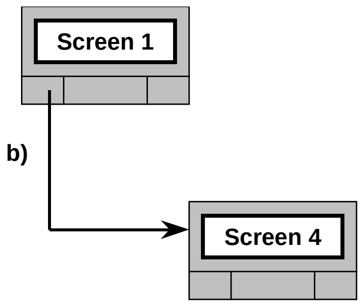


Figure 23. Screen 1 - Screen 4 -transition (b)

When a user selects the **Transaction Monitor** from Screen 1 (Figure 5) and terminal knows that there is still some transaction running, i.e. CURRENT_STATE_OF_TRANSACTION ≠ 0, then terminal displays Screen 4 (Figure 8) and offers to a user to continue active transaction or cancel it (Figure 23).

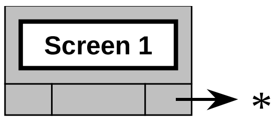


Figure 24. Screen 1 - Settings -transition

It is assumed that a user should have a possibility to adjust some settings (Table 1) for the Transaction Monitor by selecting **Settings** for the **Transaction Monitor** from Screen 1 (Figure 5). In recent level of implementation this option is not yet considered and left for the future version. Recently assume that settings are toughly installed in the Transaction Monitor (Figure 24).

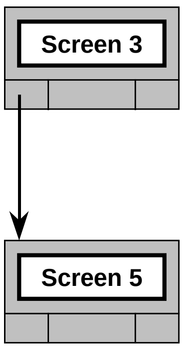


Figure 25. Screen 3 - Screen 5 -transition

When a user selects **Accept** for the **"Begin new transaction?"** query from the terminal Screen 3 (Figure 7) the terminal displays Screen 5 (Figure 9) and offers to a user to start the new transaction basing either on recent state of parameters (which was left from previous transaction) or restore and use the default settings for parameters (Figure 25).

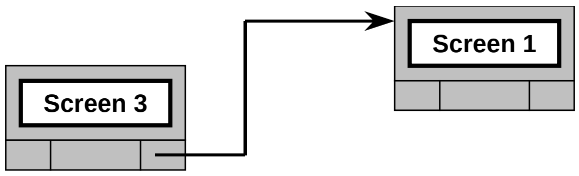


Figure 26. Screen 3 - Screen 1 -transition

Figure 26: When a user selects **Exit Monitor** for the **"Begin new transaction?"** query from the terminal Screen 3 (Figure 7) the terminal returns to initial Screen 1 (Figure 5).

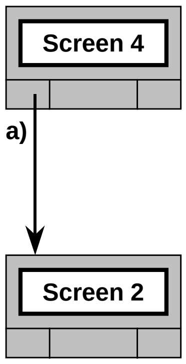


Figure 27. Screen 4 - Screen 2 -transition (a)

Assume that a user selects **Accept** for the **"You have transaction running. Continue?"** query from the terminal in Screen 4 (Figure 8). If in that state the terminal knows that a subtransaction with one of services was not yet finished (i.e. CURRENT_STATE OF_SUBTRANSACTION ≠ 0), then the terminal displays Screen 2 (Figure 6) of the service, which SERVICE_ID = CURRENT_STATE_OF_TRANSACTION, and requests from a user to login to this service (Figure 27).

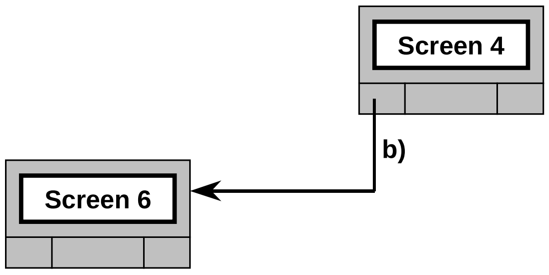


Figure 28. Screen 4 - Screen 6 -transition (b)

Assume that a user selects **Accept** for the **"You have transaction running. Continue?"** query from the terminal in Screen 4 (Figure 8). If in that state the terminal knows that there is no any unfinished (active) subtransaction with any of services (i.e. CURRENT_STATE OF_SUBTRANSACTION = 0), then the terminal displays Screen 6 (Figure 10) and offers to a user to select a service for the new subtransaction within the active transaction (Figure 28).

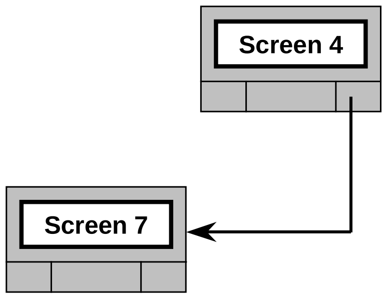


Figure 29. Screen 4 - Screen 7 -transition

If a user selects **Cancel** transaction for the **"You have transaction running. Continue?"** query from the terminal in Screen 4 (Figure 8), then he will receive a warning from the transaction monitor (Screen 7, Figure 11) with the request to confirm cancellation (Figure 29).

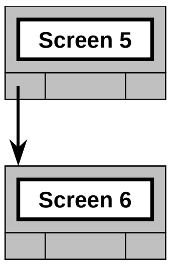


Figure 30. Screen 5 - Screen 6 -transition (left)

If a user selects **Accept** for the **"Use parameters from the previous transaction?"** query from the terminal in Screen 5 (Figure 9), then without any changes in the parameters the terminal switches to Screen 6 (Figure 10) and offers to a user to select a service for the new subtransaction (Figure 30).

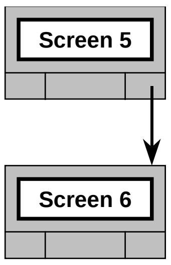


Figure 31. Screen 5 - Screen 6 -transition (right)

If a user selects **Use defaults** for the **"Use parameters from the previous transaction?"** query from the terminal in Screen 5 (Figure 9), then the terminal restores default settings for the parameters (i.e. assigns PARAMETER_RECENT_VALUE = PARAMETER_DEFAULT_VALUE and PARAMETER _CANCEL_SUBTRANSACTION_VALUE = PARAMETER_DEFAULT_ VALUE) and switches to Screen 6 (Figure 10) offering to a user to select a service for the new subtransaction (Figure 31).

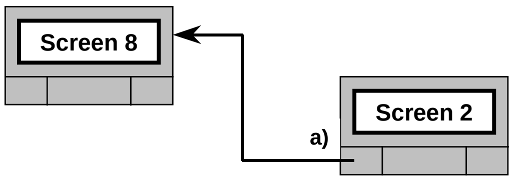


Figure 32. Screen 2 - Screen 8 -transition (a)

If a user successfully logins (i.e. the response for the LOGIN query contains LOGIN_REPLY = 'OK', Figure 4) to the service with appropriate SERVICE_ID in Screen 2 (Figure 6) and CURRENT_STATE_OF_SUBTRANSACTION = 0, then the terminal assigns CURRENT_ STATE_OF_TRANSACTION = SERVICE_ID, and displays Screen 8 (Figure 12), where the service asks a user whether to begin a new subtransaction (Figure 32).

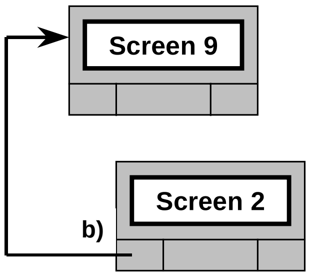


Figure 33. Screen 2 - Screen 9 -transition (b)

If a user successfully logins to the service in Screen 2 (Figure 6) and we have: CURRENT_STATE_OF_SUBTRANSACTION ≠ 0 and ATOMICITY_PROTECTOR = 0, then the terminal displays Screen 9 (Figure 13), where the Transaction Monitor informs a user that there is still an active subtransaction and asks whether to continue it (Figure 33).

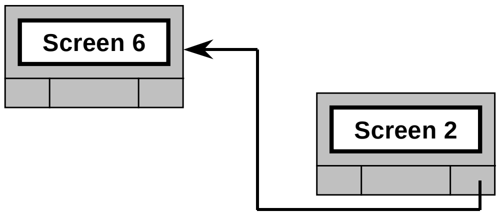


Figure 34. Screen 2 - Screen 6 -transition

If a user cancels login action or fails to login after 3 trials (i.e. the response for the LOGIN query contains LOGIN_REPLY = 'FAIL' during three trials) in Screen 2 (Figure 6), then the terminal assigns ATOMICITY_PROTECTOR = 0, voluntary ends subtransaction, which was active (i.e. assigns CURRENT_STATE_OF_ SUBTRANSACTION = 0), and displays Screen 6 (Figure 10) offering to a user to select another service for a new subtransaction (Figure 34).

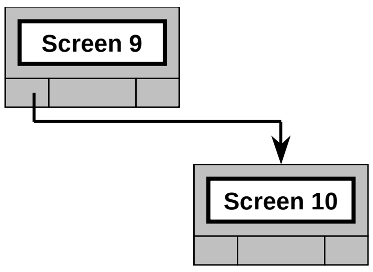


Figure 35. Screen 9 - Screen 10 -transition

If a user selects **Accept** for the **"You have subtransaction running. Continue?"** query from the terminal in Screen 9 (Figure 13), then the terminal displays Screen 10 (Figure 14) of the service, and asks a user to select next action from the LIST_OF_AVAILABLE_ACTIONS (Figure 35).

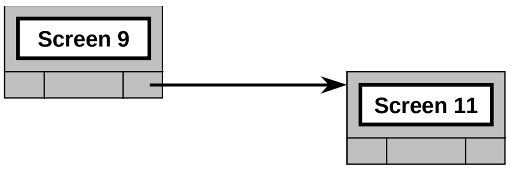


Figure 36. Screen 9 - Screen 11 -transition

If a user selects **Cancel Subtransaction** for the **"You have subtransaction running. Continue?"** query from the terminal in Screen 9 (Figure 13), then the terminal displays Screen 11 (Figure 15) of the service with a warning from the service requesting to confirm cancellation (Figure 36).

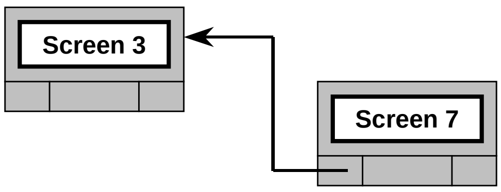


Figure 37. Screen 7 - Screen 3 -transition

Figure 37: If a user after receiving warning **"This will cancel an active transaction!"** (Screen 7, Figure 11) nevertheless confirms cancellation, then the terminal assigns ATOMICITY_ PROTECTOR = 0, CURRENT_STATE_OF_TRANSACTION = 0 and suggests to a user to begin a new transaction (Screen 3, Figure 7).

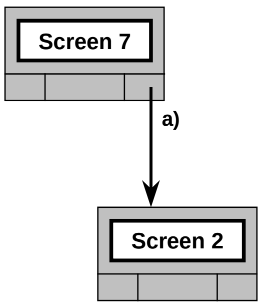


Figure 38. Screen 7 - Screen 2 -transition (a)

If a user after receiving warning **"This will cancel an active transaction!"** (Screen 7, Figure 11) changes his mind and decides to continue an active transaction and CURRENT_STATE_OF_ SUBTRANSACTION ≠ 0, then the terminal displays Screen 2 (Figure 6) of the service, which SERVICE_ID = CURRENT_STATE_OF_TRANSACTION, and requests from a user to login to this service (Figure 38).

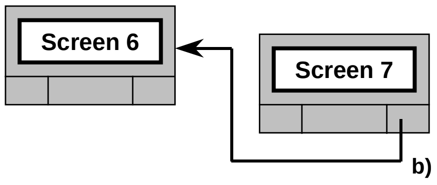


Figure 39. Screen 7 - Screen 6 -transition

If a user after receiving warning **"This will cancel an active transaction!"** (Screen 7, Figure 11) changes his mind and decides to continue an active transaction and CURRENT_STATE_OF_ SUBTRANSACTION = 0, then the terminal switches to Screen 6 (Figure 10) offering to a user to select a service for the new subtransaction (Figure 39).

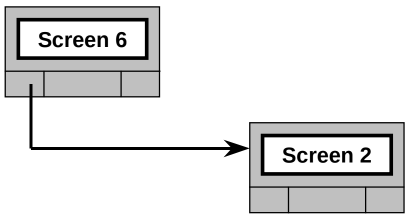


Figure 40. Screen 6 - Screen 2 -transition

When a user selects the service from Screen 6 (Figure 10), the terminal displays Screen 2 (Figure 6) with this selected service SEVICE_ID and requests from a user to login to this service (Figure 40).

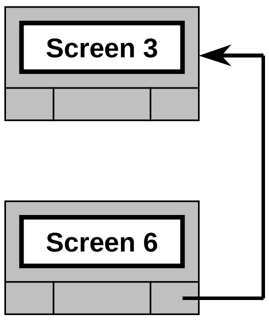


Figure 41. Screen 6 - Screen 3 -transition

If a user selects option **End transaction** from Screen 6 (Figure 10), then the terminal assigns CURRENT_STATE_OF_TRANSACTION = 0 and switches to Screen 3 (Figure 7) suggesting to a user to begin a new transaction (Figure 41).

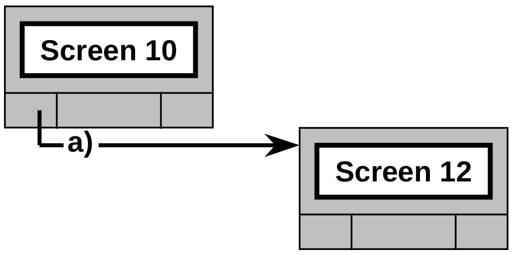


Figure 42. Screen 10 - Screen 12 -transition (a)

In Screen 10 (Figure 14) a user selects one of the LIST_OF_AVAILABLE_ACTIONS in the service. Based on selection the terminal assigns ACTIVE_ACTION_ID (selection among possible choices of the **End subtransaction**, i.e. LOGOUT will be the next case). If a user selects some action, which is not **End transaction** action, then the terminal comes to Screen 12 (Figure 16) of the action, which ID = ACTIVE_ACTION_ID, displays and offers to a user to edit INPUT_PARAMETERS_FOR _ACTION (Figure 42).

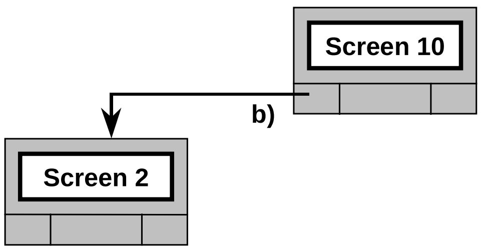


Figure 43. Screen 10 - Screen 2 -transition (b)

If in Screen 10 (Figure 14) a user selects the **End subtransaction** action from the LIST_OF_ AVAILABLE_ACTIONS, then the terminal completes LOGOUT action, which assigns ACTIVE_ACTION_ID = 0, CURRENT_STATE_OF_SUBTRANSACTION = 0 and switches to Screen 2 (Figure 6) offering to a user to login for a new subtransaction (Figure 43).

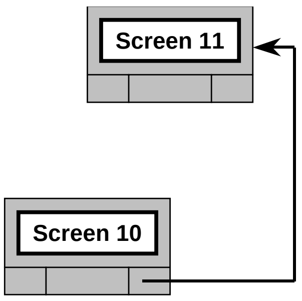


Figure 44. Screen 10 - Screen 11 -transition

If in Screen 10 (Figure 14) a user selects the **Cancel subtransaction** option, then the terminal displays Screen 11 (Figure 15) with warning **"This will cancel active subtransaction"** and asks a user to confirm cancellation (Figure 44).

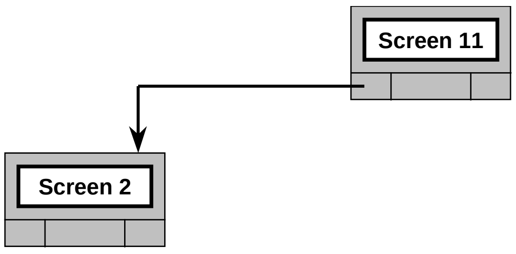


Figure 45. Screen 11 - Screen 2 -transition

If in Screen 11 (Figure 15) a user confirms the cancellation of active subtransaction, then the terminal restores PARAMETER_CANCEL_SUBTRANSACTION_VALUE for each parameter (assigns PARAMETER_RECENT_VALUE = PARAMETER_CANCEL_ SUBTRANSACTION_ VALUE), assigns CURRENT_STATE_OF_SUBTRANSACTION = 0, makes LOGOUT action, which assigns ACTIVE_ACTION_ID = 0, CURRENT_STATE_OF_SUBTRANSACTION = 0 and switches to Screen 2 (Figure 6) offering to a user to login for a new subtransaction (Figure 43).

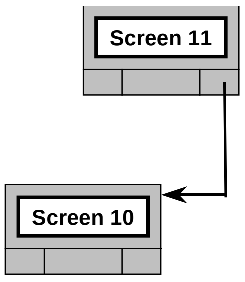


Figure 46. Screen 11 - Screen 10 -transition

If in Screen 11 (Figure 15) a user refuses to confirm the cancellation of active subtransaction and decides to continue it, then the terminal switches to Screen 10 (Figure 14) suggesting to a user to select the next action for the active subtransaction (Figure 46).

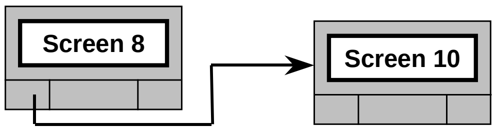


Figure 47. Screen 8 - Screen 10 -transition

In Screen 8 (Figure 12) the Monitor shows the BILL_RECENT_VALUE for the use of appropriate Service and asks whether to begin a new subtransaction. If a user selects **Accept** for the **"Begin new subtransaction?"** query from the terminal in Screen 8, then the terminal assigns, PARAMETER_CANCEL_SUBTRANSACTION_VALUE = PARAMETER_RECENT_VALUE for each parameter, assigns CURRENT_STATE_OF_SUBTRANSACTION = 1 (after-login state or S_1 in Figure 2), compiles and submits appropriate service query (Figure 3-a) to the service, assigns ATOMICITY_PROTECTOR = 1, receives response from service (Figure 3-b), assigns ATOMICITY_PROTECTOR = 0, displays Screen 10 (Figure 14) of the service, and asks from a user to select an action from the delivered LIST_OF_AVAILABLE_ACTIONS (Figure 47).

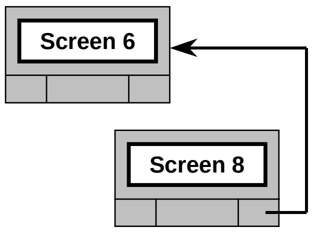


Figure 48. Screen 8 - Screen 6 -transition

If a user selects **Exit Service** for the **"Begin new subtransaction?"** query from the terminal in Screen 8 (Figure 12), then the terminal switches to Screen 6 (Figure 10) suggesting to a user to select another service to continue active transaction (Figure 48).

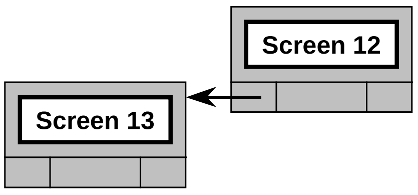


Figure 49. Screen 12 - Screen 13 -transition

In Screen 12 (Figure 16) a user is offered to select a parameter of the action from the list of INPUT_PARAMETERS_FOR_ACTION, which a user wants to edit before performing the action. If a user selects one of such parameters, then the terminal appropriately assigns the ACTIVE_PARAMETER_ID and switches to Screen 13 (Figure 17) where a user can accept displayed value (PARAMETER_RECENT_VALUE) of the parameter, enter any other value or use PARAMETER_DEFAULT_VALUE for the parameter (Figure 49).

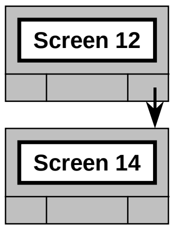


Figure 50. Screen 12 - Screen 14 -transition

If in Screen 12 (Figure 16) a user selects **Submit query** option for the appropriate action, then the terminal completed the query using PARAMETER_RECENT_VALUE for each parameter from the list of INPUT_PARAMETERS_FOR_ACTION according to the format from Figure 3-a, submits it to the service assigning ATOMICITY_PROTECTOR = 1 and, when the appropriate response (Figure 3-b) from the service arrives, the terminal switches to Screen 14 (Figure 18) requesting a user to accept the delivery (Figure 50).

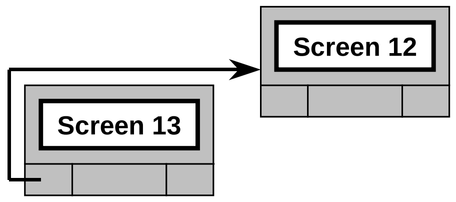


Figure 51. Screen 13 - Screen 12 -transition

If in Screen 13 (Figure 17) a user accepts displayed value (PARAMETER_RECENT_VALUE) of the active parameter, or enters other value (i.e. changes PARAMETER_RECENT_VALUE appropriately) then the terminal returns to Screen 12 (Figure 16) offering to select another parameter for editing from the list of INPUT_PARAMETERS_FOR_ACTION (Figure 51).

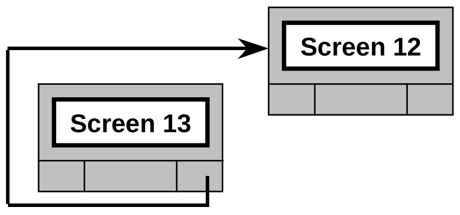


Figure 52. Screen 13 - Screen 12 -transition

If in Screen 13 (Figure 17) a user selects **Use default** option for the active parameter, then the terminal assigns PARAMETER_RECENT_VALUE = PARAMETER_DEFAULT_VALUE for this parameter and returns to Screen 12 (Figure 16) offering to select another parameter for editing from the list of INPUT_PARAMETERS_FOR_ACTION (Figure 52).

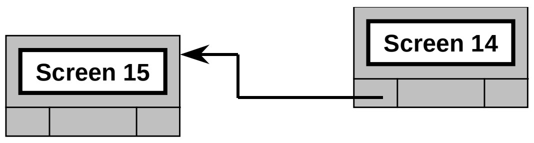


Figure 53. Screen 14 - Screen 15 -transition

In Screen 14 (Figure 18) the terminal displays the delivery report from the service concerning the results of the last submitted query and the PRICE_FOR_LAST_ACTION value for the delivered data. If a user accepts the result and pricing of the delivery, then the terminal increments the user's appropriate bill (i.e. bill for the service, which ID = CURRENT_STATE_OF_TRANSACTION) by the following way: BILL_RECENT_VALUE = BILL_RECENT_VALUE + PRICE_FOR_LAST_ ACTION, makes appropriate changes in parameters (i.e. accepts delivered values of parameters from the list of OUTPUT_PARAMETERS_FROM_ACTION as PARAMETER_RECENT_ VALUE for each of delivered parameters), accepts and reassigns the LIST_OF_AVAILABLE_ ACTIONS according to delivered one and switches to Screen 15 (Figure 19) giving to a user access to the list of OUTPUT_PARAMETERS_FROM_ACTION (Figure 53).

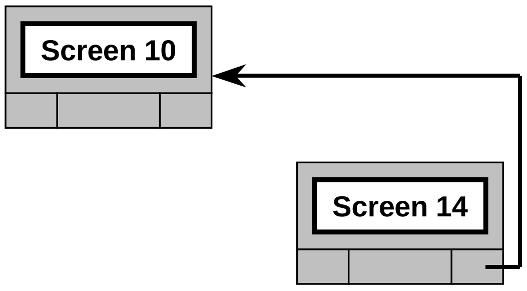


Figure 54. Screen 14 - Screen 10 -transition

If a user refuses to accept the result of the delivery from Screen 14 (Figure 18), i.e. selects **Cancel delivery** option then the terminal lefts the user's appropriate bill (BILL_RECENT_VALUE) and appropriate parameters from the list of OUTPUT_PARAMETERS_FROM_ACTION without changes, sends to the service CANCEL_RESPONSE query (Figure 3-e), receives RESPONSE_CANCELLED response from the service and switches to Screen 10 (Figure 14) asking a user to select another action. The service when receives CANCEL_RESPONSE query should exclude the PRICE_FOR_LAST_ACTION value from a user's BILL_RECENT_ VALUE, which service has previously incremented by default (Figure 54).

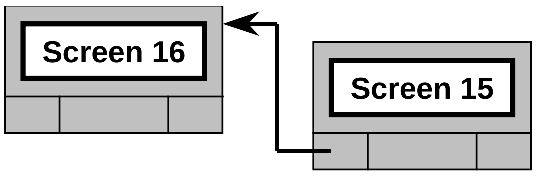


Figure 55. Screen 15 - Screen 16 -transition

In Screen 15 (Figure 19) the list of OUTPUT_PARAMETERS_FROM_ACTION from the last performed action (which ID is equal to ACTIVE_ACTION_ID) is displayed. If a user selects one of them for **View**, then the terminal appropriately assigns the ACTIVE_PARAMETER_ID and switches to Screen 16 (Figure 20) where a user can left the delivered value as the PARAMETER_ RECENT_VALUE, change value or restore the PARAMETER_DEFAULT_VALUE for this active parameter (Figure 55).

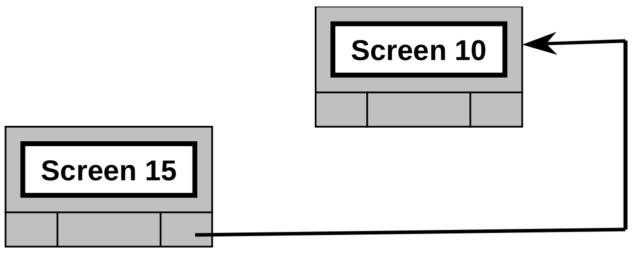


Figure 56. Screen 15 - Screen 10 -transition

In Screen 15 (Figure 19) the list of OUTPUT_PARAMETERS_FROM_ACTION is displayed. If a user does not want to view them and selects **Back to subtransaction** option then the terminal switches to Screen 10 (Figure 14) asking a user to select another action from the LIST_OF_AVAILABLE_ ACTIONS within an active subtransaction (Figure 56).

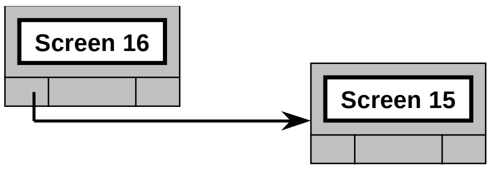


Figure 57. Screen 16 - Screen 15 -transition (left)

If in Screen 16 (Figure 20) a user accepts displayed value (PARAMETER_RECENT_VALUE) of the active parameter, or enters other value (i.e. changes PARAMETER_RECENT_VALUE appropriately) then the terminal returns to Screen 15 (Figure 19) offering to select another parameter from the list of OUTPUT_PARAMETERS_FROM_ACTION for viewing/editing (Figure 57).

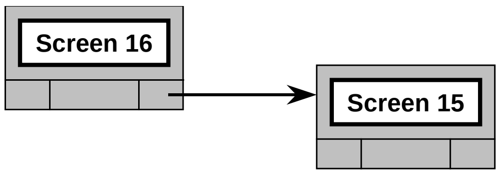


Figure 58. Screen 16 - Screen 15 -transition (right)

If in Screen 16 (Figure 20) a user selects **Use default** option for the active parameter, then the terminal assigns PARAMETER_RECENT_VALUE = PARAMETER_DEFAULT_VALUE for this parameter and returns to Screen 15 (Figure 19) offering to select another parameter from the list of OUTPUT_PARAMETERS_FROM_ACTION for viewing/editing (Figure 58).

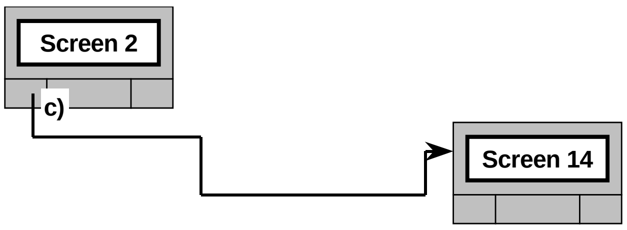


Figure 59. Screen 2 - Screen 14 -transition (c)

If a user successfully logins to the service in Screen 2 (Figure 6) and we have: CURRENT_STATE_OF_ SUBTRANSACTION $\neq$ 0 and ATOMICITY_PROTECTOR = 1, then the terminal submits the query REPEAT_RESPONSE (according the format from Figure 3-c) to the service and, when the appropriate response (Figure 3-d) from the service arrives, the terminal assigns ATOMICITY_PROTECTOR = 0 and switches to Screen 14 (Figure 18) requesting a user to accept the delivery (Figure 59).

4 Ontology-based Transaction Monitor for m-commerce location-based services

Here we consider the implementation of the ontology-based Transaction Monitor to manage transactions for location-based services. We consider the model example of two services, LBS and positioning service, across which the Transaction Monitor will perform transactions. For that we will define necessary model ontologies and show basic stages of transactional process with the Monitor.

4.1 Constants and ontologies for the LBS model example

We will consider the following basic constants and the subset of parameters, actions and appropriate ontologies to describe the LBS-type services as presented in Table 5 and Figures 59-61.

Table 5. Constants and ontologies for the LBS

ID of the Constant	Value
Basic constants:	
TERMINAL_ID	From settings
TOTAL_NUMBER_OF_SERVICES	2
TOTAL_NUMBER_OF_ACTIONS	3
TOTAL_NUMBER_OF_PARAMETERS	9
SERVICE_ID [1]	POSITIONING_SERVICE
SERVICE_ID [2]	LOCATION_BASED_SERVICE
Service atomic action ontologies:	
ACTION_ID [1]	TO_LOCATE_BY_ID
ACTION_ID [2]	TO_LOCATE_BY_ADDRESS
ACTION_ID [3]	TO_GET_MAP
INPUT_PARAMETERS_FOR_ACTION [1,*]	TERMINAL_ID
OUTPUT_PARAMETERS_FROM_ACTION [1,*]	LATITUDE_WGS_84 LONGITUDE_WGS_84 ALTITUDE_WGS_84
INPUT_PARAMETERS_FOR_ACTION [2,*]	STREET_NUMBER STREET_NAME CITY_NAME STATE_OR_PROVINCE_NAME COUNTRY_NAME
OUTPUT_PARAMETERS_FROM_ACTION [2,*]	LATITUDE_WGS_84 LONGITUDE_WGS_84
INPUT_PARAMETERS_FOR_ACTION [3,*]	LATITUDE_WGS_84 LONGITUDE_WGS_84
OUTPUT_PARAMETERS_FROM_ACTION [3,*]	LOCATION_BASED_MAP

Parameter ontologies:	
PARAMETER_ID [1]	LATITUDE_WGS_84
PARAMETER_DEFAULT_VALUE [1]	0 [or optional: latitude of Jyvaskyla RW Station]
PARAMETER_TYPE/SCHEMA [1]	32-bit signed integer
PARAMETER_ID [2]	LONGITUDE_WGS_84
PARAMETER_DEFAULT_VALUE [2]	0 [or optional: longitude of Jyvaskyla RW Station]
PARAMETER_TYPE/SCHEMA [2]	32-bit signed integer
PARAMETER_ID [3]	ALTITUDE_WGS_84
PARAMETER_DEFAULT_VALUE [3]	0 [or optional: altitude of Jyvaskyla RW Station]
PARAMETER_TYPE/SCHEMA [3]	32-bit signed integer
PARAMETER_ID [4]	STREET_NUMBER
PARAMETER_DEFAULT_VALUE [4]	16
PARAMETER_TYPE/SCHEMA [4]	2-byte unsigned integer
PARAMETER_ID [5]	STREET_NAME
PARAMETER_DEFAULT_VALUE [5]	HANNIKAISENKATU
PARAMETER_TYPE/SCHEMA [5]	ASCII text
PARAMETER_ID [6]	CITY_NAME
PARAMETER_DEFAULT_VALUE [6]	JYVASKYLA
PARAMETER_TYPE/SCHEMA [6]	ASCII text
PARAMETER_ID [7]	STATE_OR_PROVINCE_NAME
PARAMETER_DEFAULT_VALUE [7]	CENTRAL_FINLAND
PARAMETER_TYPE/SCHEMA [7]	ASCII text
PARAMETER_ID [8]	COUNTRY_NAME
PARAMETER_DEFAULT_VALUE [8]	FINLAND
PARAMETER_TYPE/SCHEMA [8]	ASCII text
PARAMETER_ID [9]	LOCATION_BASED_MAP
PARAMETER_DEFAULT_VALUE [9]	Map around Jyvaskyla Railway Station
PARAMETER_TYPE/SCHEMA [9]	GMML file

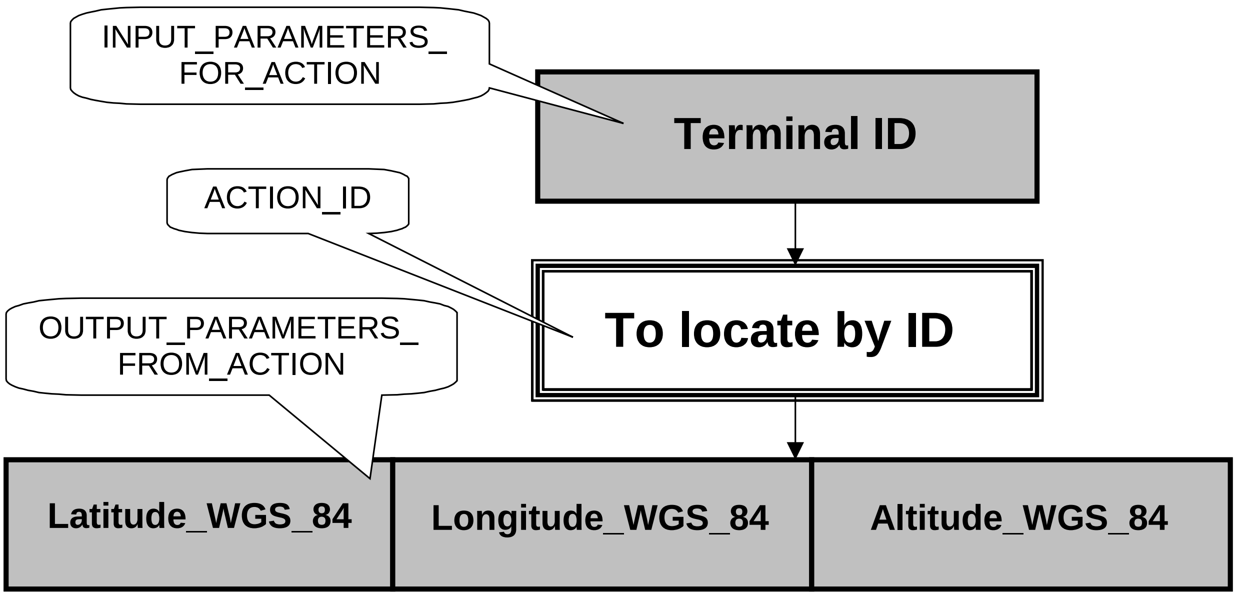


Figure 59. Ontology for the "TO_LOCATE_BY_ID" action

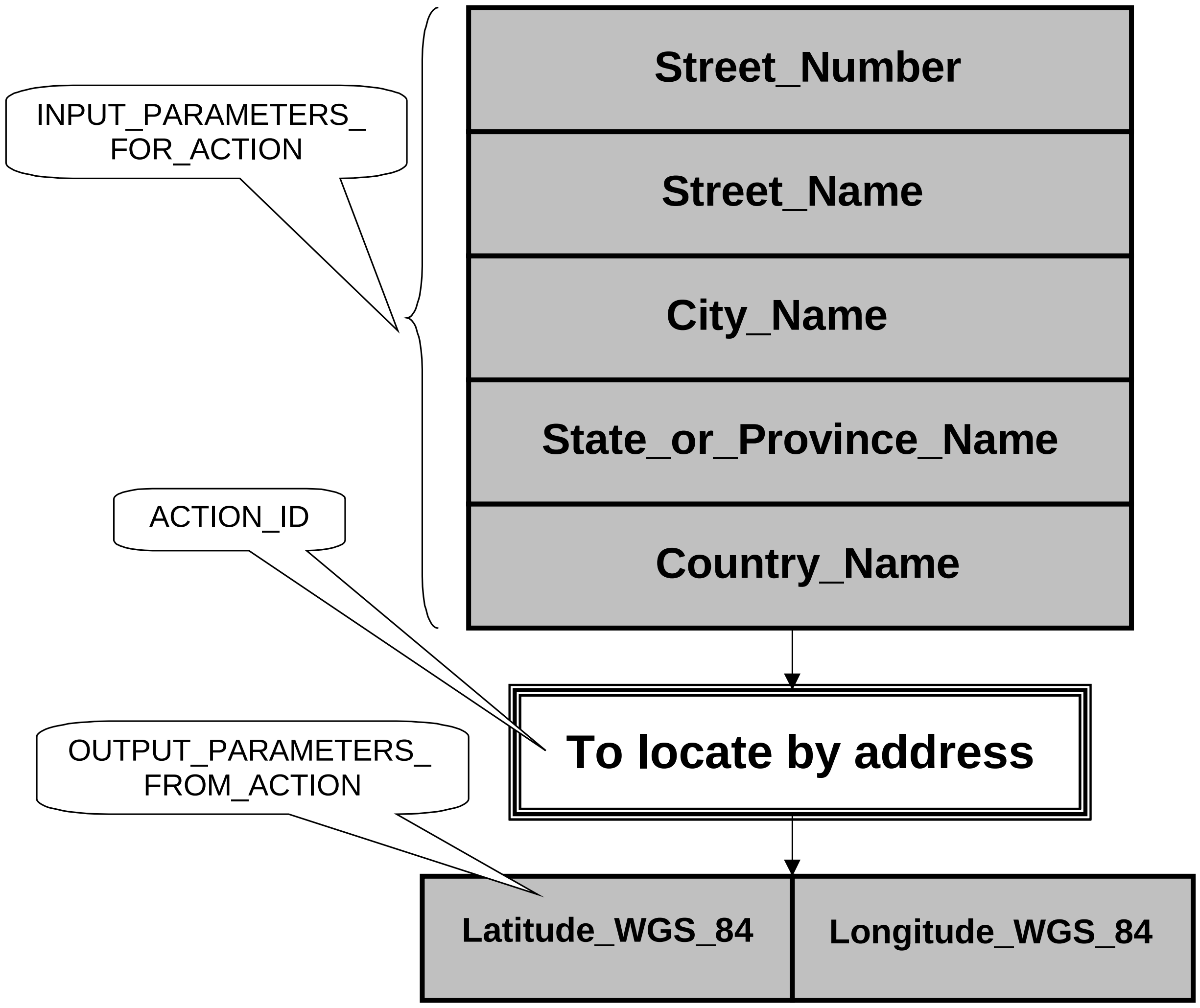


Figure 60. Ontology for the "TO_LOCATE_BY_ADDRESS" action

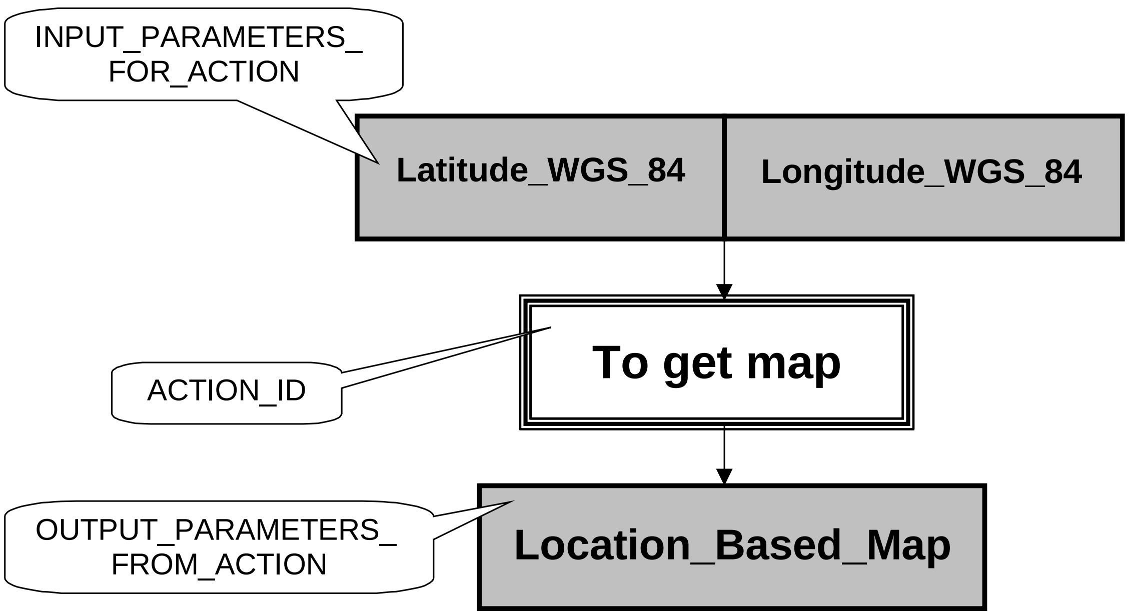


Figure 61. Ontology for the "TO_GET_MAP" action

4.2 Service trees for the LBS model example

We are considering two services: positioning service and location-based service (LBS). Suppose that the positioning service performs only one simple action - locating the mobile terminal based on its ID. The location-based service can also return to a user its location but for that the LBS itself makes query to positioning service to get the location. However we assume that our model LBS can also transfer submitted street address to the coordinates. Also, based on location, the LBS can deliver an appropriate Map to a user's terminal. Thus the appropriate service trees for positioning service and LBS can be presented as it shown in Figures 62 and 63 respectively.

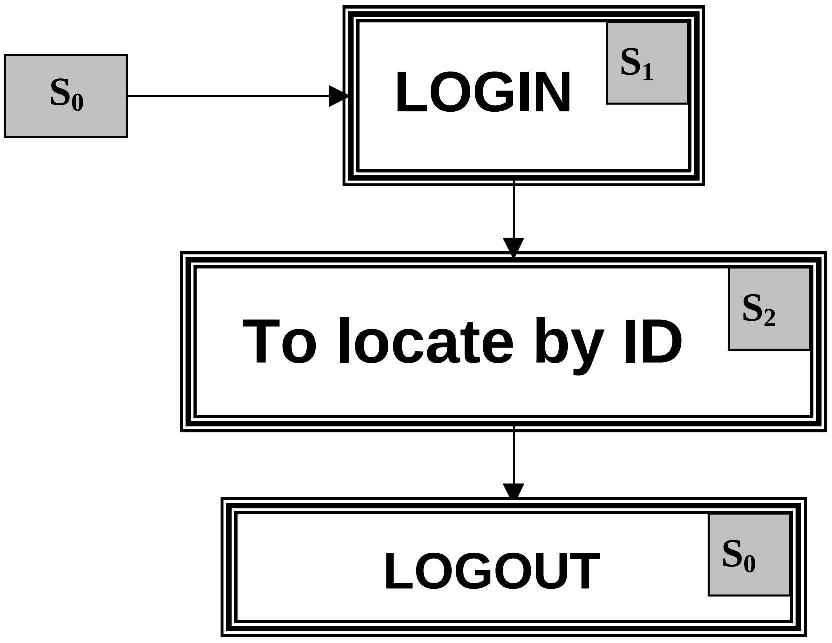


Figure 62. Service tree of the Positioning Service

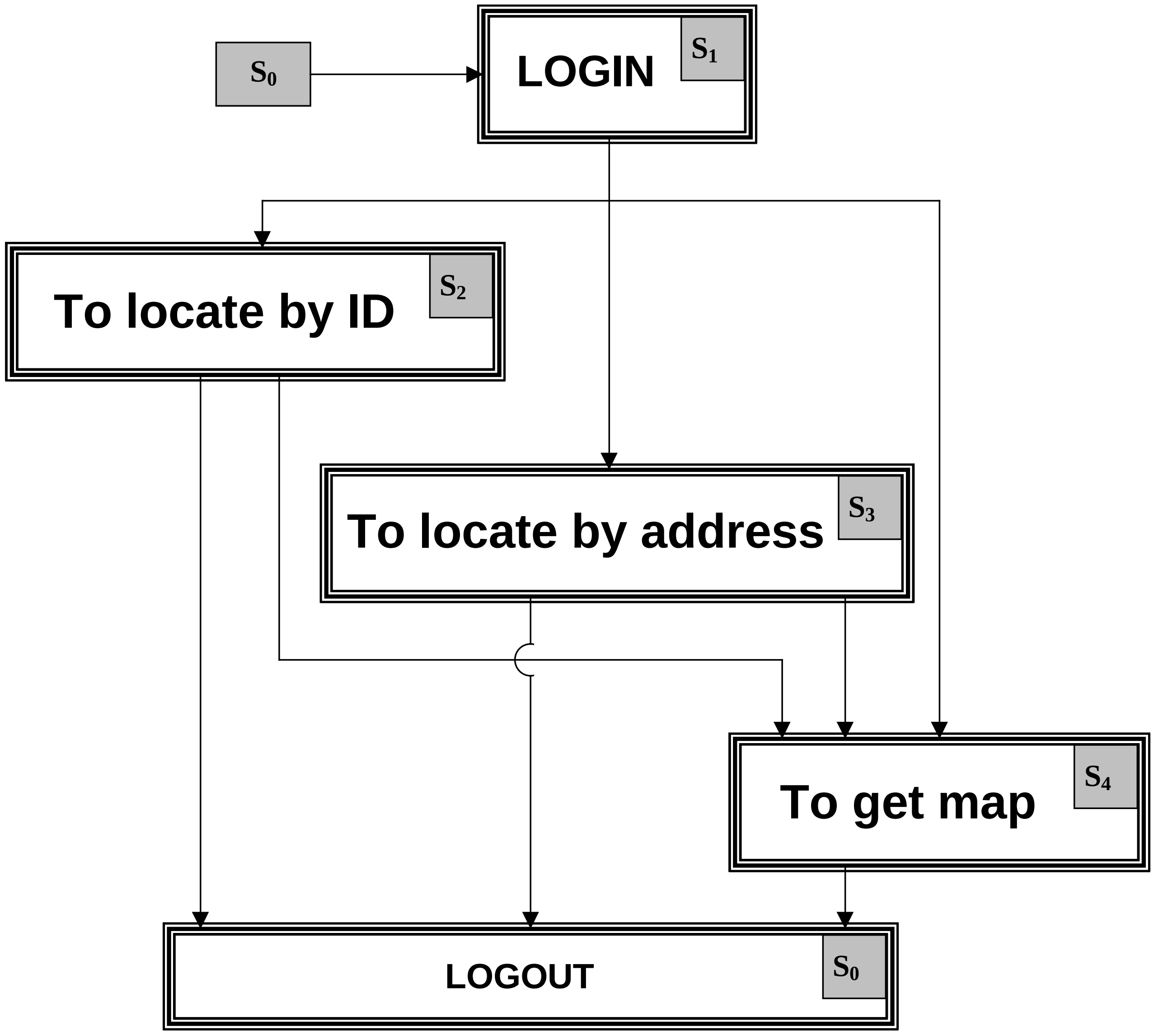


Figure 63. Service tree of the Location Based Service

4.3 Transaction sequence diagrams for the LBS model example

Assume that a user of the terminal, which is equipped by the Transaction Monitor, is a registered user of the two above-mentioned services (positioning service and LBS).

We consider a case that a user needs a map, associated with his current location. Scenarios of the use of the Transaction Monitor to get map by managing transactions across two services are shown in Figures 64 - 67 in the form of sequence diagrams.

Notice that these are the model examples and for commercial implementation these scenarios as well as Transaction Monitor functionality can be optimised and further developed.

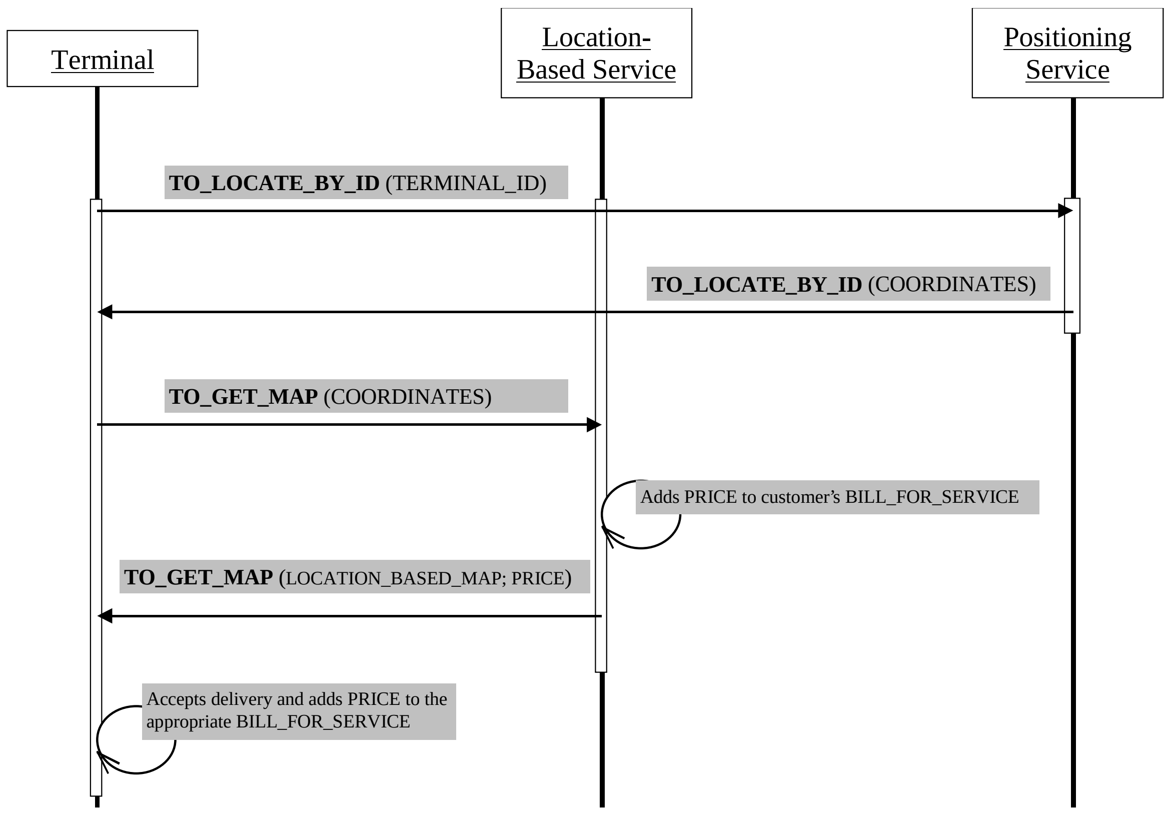

Terminal
Location-Based Service
Positioning Service
TO_LOCATE_BY_ID (TERMINAL_ID)
TO_LOCATE_BY_ID (COORDINATES)
TO_GET_MAP (COORDINATES)
Adds PRICE to customer's BILL_FOR_SERVICE
TO_GET_MAP (LOCATION_BASED_MAP; PRICE)
Accepts delivery and adds PRICE to the appropriate BILL_FOR_SERVICE

Figure 64. Case when a user locates himself and then submits coordinates to LBS

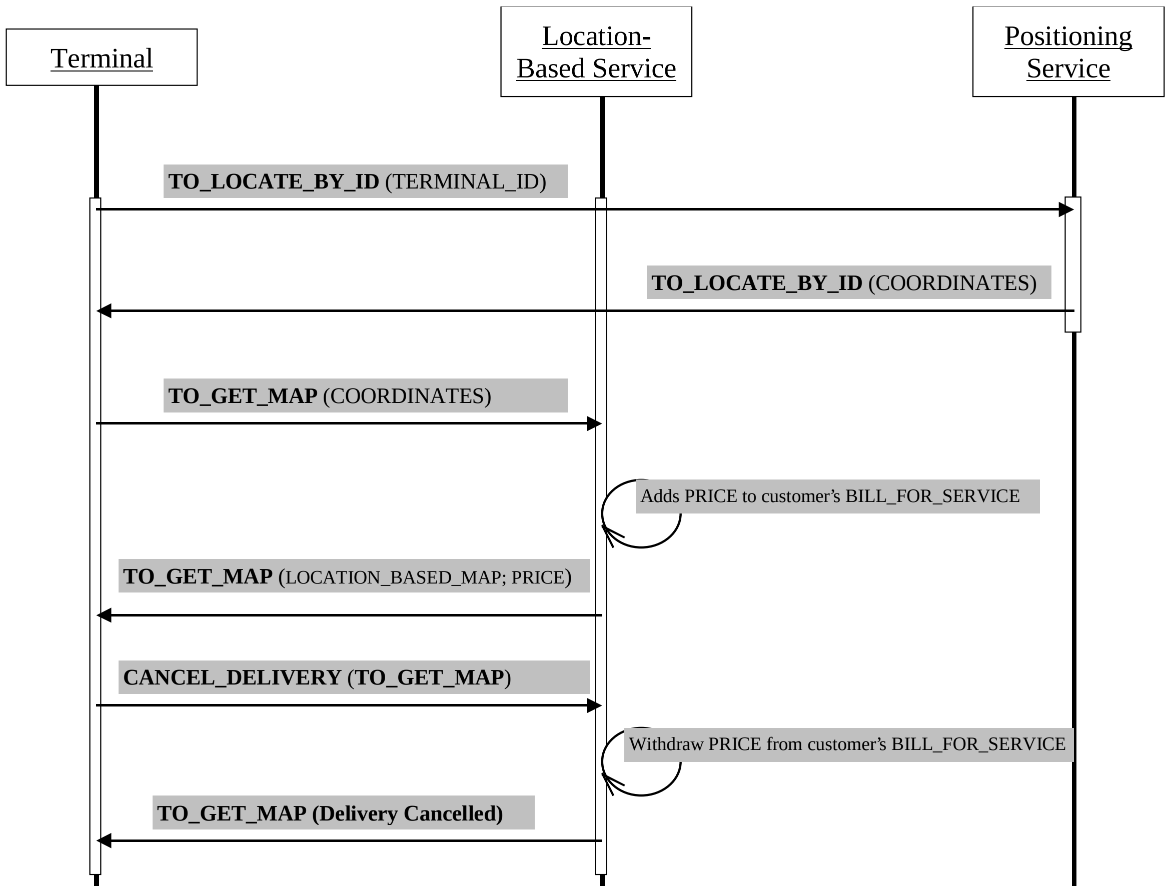

Terminal
Location-Based Service
Positioning Service
TO_LOCATE_BY_ID (TERMINAL_ID)
TO_LOCATE_BY_ID (COORDINATES)
TO_GET_MAP (COORDINATES)
Adds PRICE to customer's BILL_FOR_SERVICE
TO_GET_MAP (LOCATION_BASED_MAP; PRICE)
CANCEL_DELIVERY (TO_GET_MAP)
Withdraw PRICE from customer's BILL_FOR_SERVICE
TO_GET_MAP (Delivery Cancelled)

Figure 65. Case as in Figure 64, but a user refuses to accept the map delivery

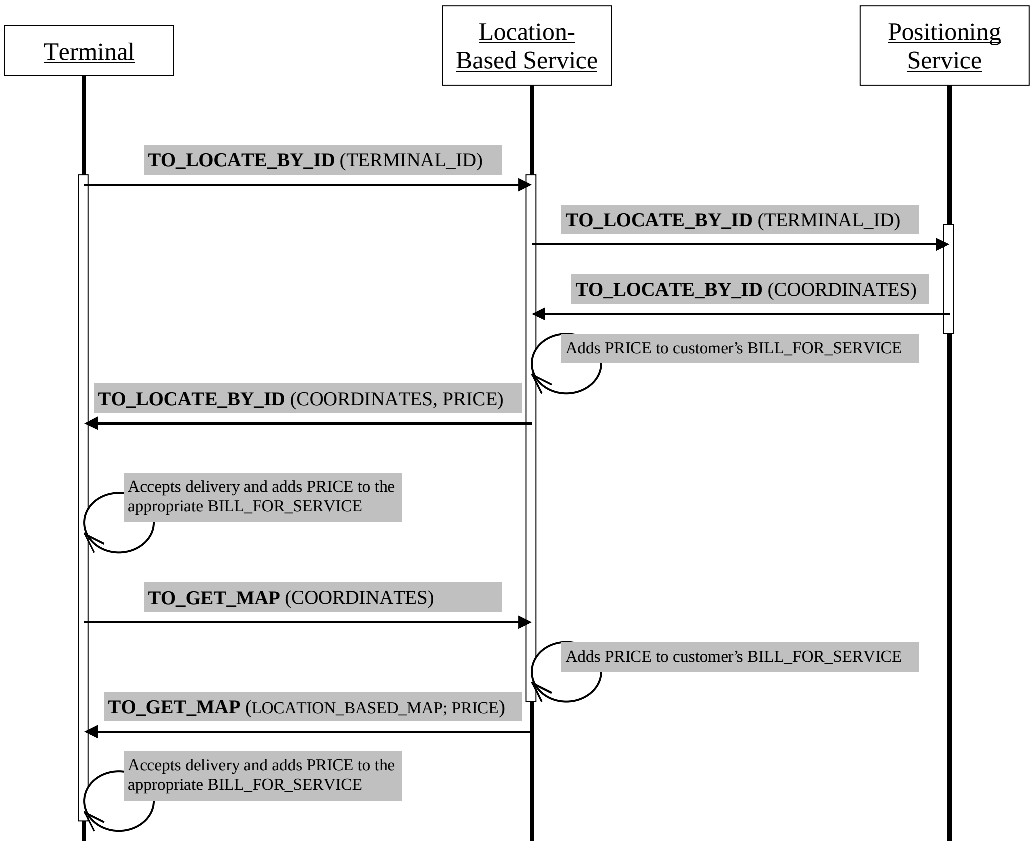

Terminal
Location-Based Service
Positioning Service
TO_LOCATE_BY_ID (TERMINAL_ID)
TO_LOCATE_BY_ID (TERMINAL_ID)
TO_LOCATE_BY_ID (COORDINATES)
Adds PRICE to customer's BILL_FOR_SERVICE
TO_LOCATE_BY_ID (COORDINATES, PRICE)
Accepts delivery and adds PRICE to the appropriate BILL_FOR_SERVICE
TO_GET_MAP (COORDINATES)
Adds PRICE to customer's BILL_FOR_SERVICE
TO_GET_MAP (LOCATION_BASED_MAP; PRICE)
Accepts delivery and adds PRICE to the appropriate BILL_FOR_SERVICE

Figure 66. Case when LBS first locates the terminal based on ID and then delivers a map

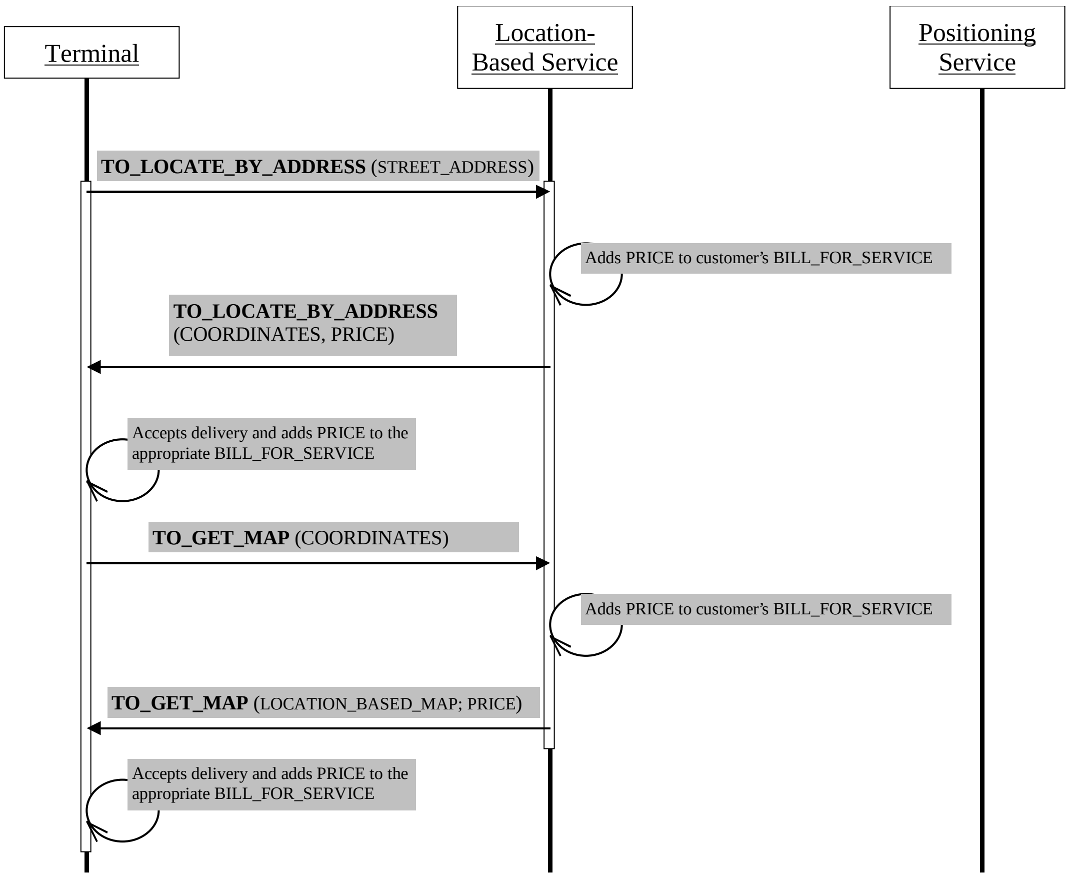

Terminal
Location-Based Service
Positioning Service
TO_LOCATE_BY_ADDRESS (STREET_ADDRESS)
Adds PRICE to customer's BILL_FOR_SERVICE
TO_LOCATE_BY_ADDRESS (COORDINATES, PRICE)
Accepts delivery and adds PRICE to the appropriate BILL_FOR_SERVICE
TO_GET_MAP (COORDINATES)
Adds PRICE to customer's BILL_FOR_SERVICE
TO_GET_MAP (LOCATION_BASED_MAP; PRICE)
Accepts delivery and adds PRICE to the appropriate BILL_FOR_SERVICE

Figure 67. LBS first locates the terminal based on street address and then delivers a map

5 Software architecture for the ontology-based Transaction Monitor

The software architecture for the Transaction Monitor consists of the following basic building blocks:

- Logger;
- Action Manager;
- Message Service;
- Bill Manager;
- Interface Manger;
- Transaction Controller;
- Subtransaction Controller.

Below there is a brief description of the building blocks for the Transaction Monitor software with appropriate modules.

5.1 Logger

Logger keeps track of the Transaction Monitor memory states in which the transaction is running. It is also responsible for the recovery of the crashed (cancelled) transactions and subtransactions by restoring appropriate parameter states.

Write_Log(PARAMETER_ID) - Assigns PARAMETER_CANCEL_SUBTRANSACTION_ VALUE = PARAMETER_RECENT_VALUE for parameter PARAMETER_ID.

Restore_Log(PARAMETER_ID) - Assigns PARAMETER_RECENT_VALUE = PARAMETER_ CANCEL_SUBTRANSACTION_VALUE for parameter PARAMETER_ID.

Restore_Default(PARAMETER_ID) - Assigns PARAMETER_RECENT_VALUE = PARAMETER_DEFAULT_VALUE for parameter PARAMETER_ID.

5.2 Action Manager

Action Manager is responsible to compiling queries from appropriate parameters of recent action and sorting responses to appropriate parameters.

Make_Query(QUERY) - compiles file QUERY from following parameters: TERMINAL_ID, CURRENT_STATE_OF_SUBTRANSACTION, ACTIVE_ACTION_ID, INPUT_PARAMETERS _FOR_ACTION.

Sort_Response(RESPONSE) - sorts file RESPONSE to following parameters: SERVICE_ID, ACTIVE_ACTION_ID, OUTPUT_PARAMETERS_FROM_ACTION, CURRENT_STATE_OF_ SUBTRANSACTION, LIST_OF_AVAILABLE_ACTIONS.

5.3 Message Service

Message Service takes care of sending queries to a service and receiving responses from it.

Send_Query(QUERY, ADDRESS_TO) - sends file QUERY to the address ADDRESS_TO.

Get_Response(ADDRESS_FROM, RESPONSE) - gets file RESPONSE from the address ADDRESS_FROM.

5.4 Bill Manager

Bill Manager is responsible for managing billing information for all services used by the Transaction Monitor

Increment_Bill - adds value PRICE_FOR_LAST_ACTION to the BILL_RECENT_VALUE of the service, which ID is equal to CURRENT_STATE_OF_TRANSACTION.

5.5 Interface Manager

Interface Manager is responsible for visualizing basic screen frames, inserting and visualizing appropriate data, by which frames are filled.

Base_for_Screen(SCREEN_FRAME) - makes SCREEN file based on appropriate SCREEN_ FRAME.

Insert_to_Screen (PLACE_ID, PARAMETER_ID) - inserts to SCREEN file to the empty slot PLACE_ID the value of the parameter PARAMETER_ID.

Show_Screen - displaying SCREEN file at the terminal screen.

Edit_Screen(PLACE_ID, PARAMETER_ID) - gets value manually edited at slot PLACE_ID of the SCREEN file as PARAMETER_RECENT_VALUE for the parameter PARAMETER_ID.

5.6 Transaction Controller

Transaction Controller is responsible for managing transactional key points like beginning, ending, canceling, and interruption. It uses functions: **Begin_Transaction**, **End_Transaction**, **Cancel_Transaction**, and **Continue_Transaction** in the way as they were described in the algorithm from 3.6.

5.7 Subtransaction Controller

Subtransaction Controller is responsible for managing subtransactional key points like beginning, ending, canceling, and interruption. It uses functions: **Begin_Subtransaction**, **End_ Subtransaction**, **Cancel_Subtransaction**, **Continue_Subtransaction,** and **Cancel_Delivery** in the way as they were described in the algorithm from 3.6.

6 Transaction atomicity protection with the ontology-based Transaction Monitor

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 [1].

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 [10].

In [12] the atomicity issues from e-commerce context were expanded to m-commerce 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 68, 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.

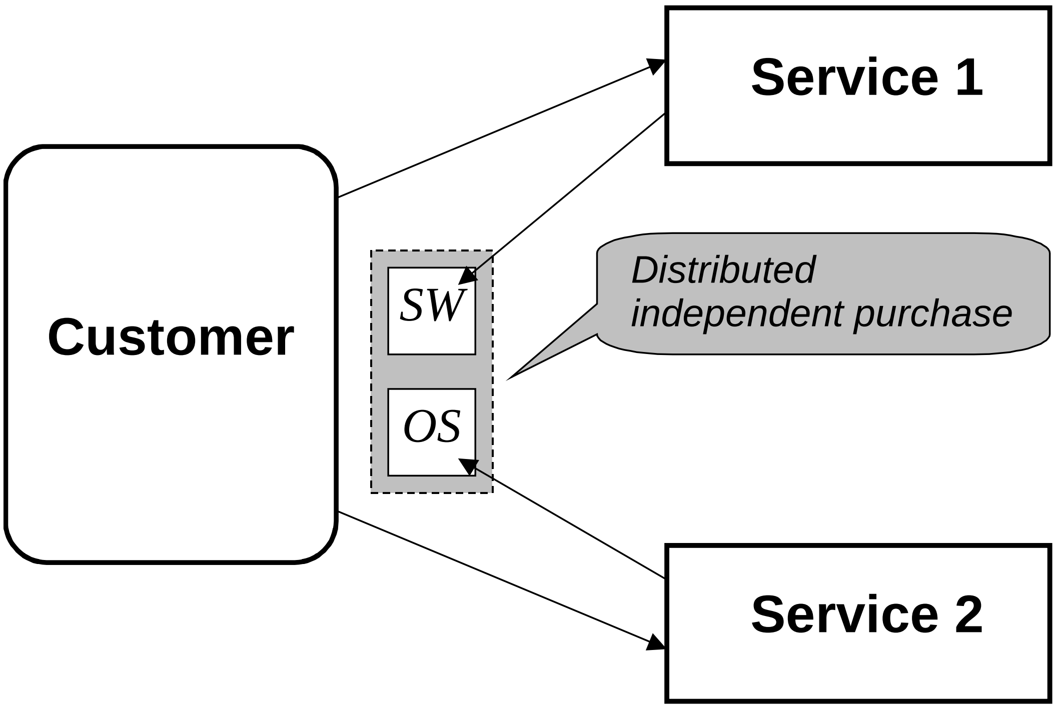


Figure 68. Distributed Purchase case where both parts of the target good can be purchased independently

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 [7] where authors apply transactional process management to realise an E-Commerce Payment Co-ordinator.

At the example in Figure 68 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 [10] as Transaction Service, or as above [7] 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 69.

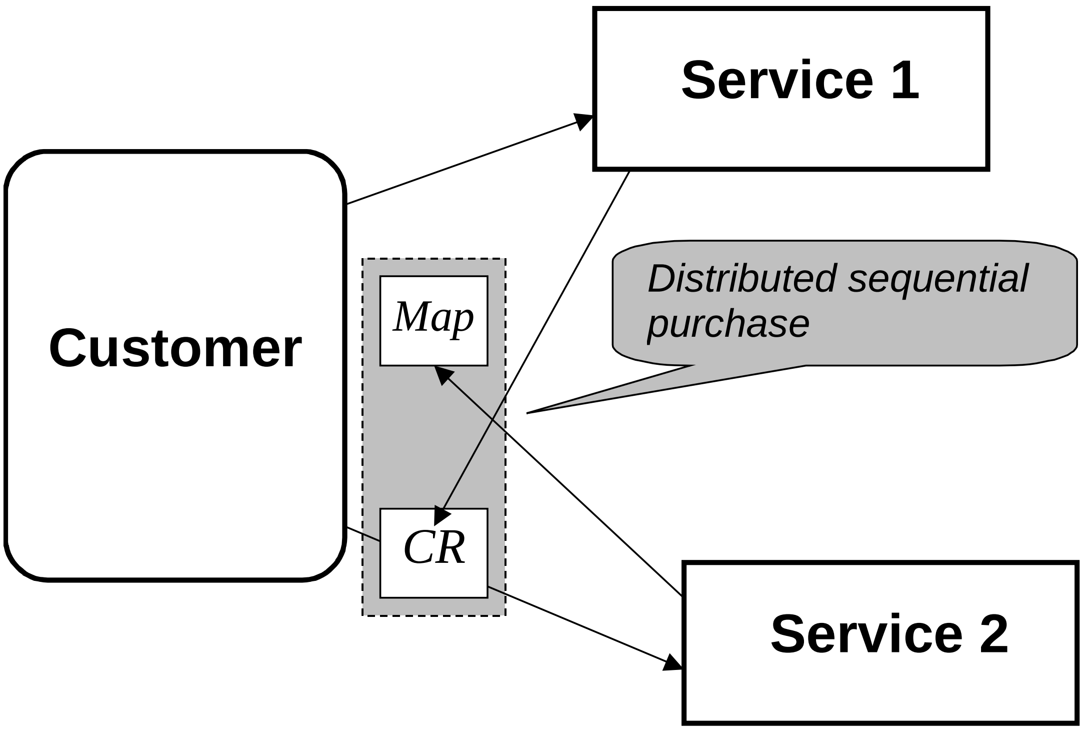


Figure 69. Distributed Purchase case where goods from distributed services should be purchased in a certain sequence

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 [10]. 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.

We are using Logger functions (see 5.1) for the recovery of crashed (cancelled) transactions and subtransactions based on roll-back technique.

7 Related work

7.1 Connection to MeT Consistent User Experience

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 Transaction Monitor, due to implementation of the concept of ontology-based transaction management, offers such a consistent standardize interface of a user with multiple services.

7.2 Connection to Hewlett Packard's E-Speak platform

E-speak is an open software platform designed specifically for the development, deployment, intelligent interaction, and management of globally distributed e-services. It allows the building of loosely coupled, distributed e-services [13]. E-Speak makes services capable to interact with each other on behalf of their users, and compose themselves into more complex services. E-speak software provides the ecosystem within which the whole sequence of transactions needed for such globally distributed and delivered services can take place - from service request, to discovery (sifting through millions of sites), to mediation (weeding out unqualified providers), to composition (mixing and matching services to fit requests), and to final delivery. The E-Speak Service Engine [13] actually is a transaction monitor software that performs the intelligent interaction of e-services.

The Transaction Monitor in the hands of user is a good example of an E-Speak engine, which expands the E-Speak internet-based framework to wireless.

7.3 Connection to the OntoWeb network platform

OntoWeb [6] - Ontology-based information exchange for knowledge management and electronic commerce is the IST Project Thematic Network of 64 academic and industrial partners inside and outside Europe. 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 such as those based on RDF and XML. Europe's cultural diversity and multi linguality, together with the strong scientific competencies existing in the ontology field, may give Europe a unique opportunity to fully exploit ontology-based technology and to play a leading role in these emerging area. The network strengthen Europe's skill base for the digital economy, maximising the impact of this key action through broad, yet cost-effective dissemination activities, activities aimed at building industrial consensus and promoting standardisation efforts in e-commerce.

The implementation of the concept of ontology-based transaction management for mobile terminal allows expanding a target for the OntoWeb framework also to m-commerce.

7.4 Connection to the *mediators and wrappers* framework

According to [2] *wrapper w* is 3-tuple *w = (ES, Q, D)*. The components of *w* are:

- an export schema *ES*;
- a set *Q* of queries against *ES* that can be executed by *w*;
- the data source *D* wrapped by *w*.

According to [2] a *mediator m* is a 3-tuple *m = (S, W, C)* and the components are:

- the mediator schema *S*;
- a set *W* of wrappers used by *m*;
- a set *C* of correspondences.

Wrappers and mediators can be considered as useful addition to the ontology-based framework in cases when some existing service cannot be easily adapted to the ontology of some e-services community.

Wrapper for a Transaction Monitor can be considered as an interface to each e-service, which defines the service schema. Mediator can be considered as an interface to a group of e-services, which defines a global schema from the local schemas and combines the schemas and the information of the local e-services.

8 Future of the ontology-based Transaction Monitor further development

The Transaction Monitor pilot implementation was restricted by certain functionality, which was decided as enough for testing and further research experiments. However we keep in mind possibilities for further development of the Transaction Monitor functionality and some possible direction of this future development are as follows:

1. *Save/Download transaction*. These functions allow to a user, after performing once some transaction with multiple services and doing routine manual work of managing it, to save this transaction with certain name for further use. When downloading the saved transaction, the monitor will perform most of previously manual actions automatically across multiple services.

2. *Save/Download parameters*. This allow to a user once obtaining some value for a parameter from a services, save it and reuse it when appropriate in many next transactions and subtransactions as input for appropriate queries.

3. *Settings management*. This might be important to flexibly add new services, create new ontologies or edit old ones when standards are being updated.

4. *Transactions security management*. This will include authentication, authorization and encryption phases to the vulnerable states of the transactional process.

5. *Transactions optimization*. This will include possibility to exclude some unnecessary actions (e.g. in Figure 66) targeting on main transactional (subtransactional) goals of a user.

9 Application-driven Transaction Monitor and its implementation plan

9.1 Basic concepts and architecture

The basic idea of the Application-Driven Transaction Monitor is to protect mobile terminal-based business applications from the unexpected disconnections and guarantee basic transactional properties for the applications. Transaction Monitor will be run in this case under control of the application and will store in a secure way and fully recover the application data if necessary whenever the interruption occurs.

In Figure 70 the general architecture of Application-Driven Transaction Monitor is presented. Application will communicate with Transaction Monitor via Dispatcher, which further distribute all application queries among two basic modules: Restorator and Archivist, collects responses and delivers to Application.

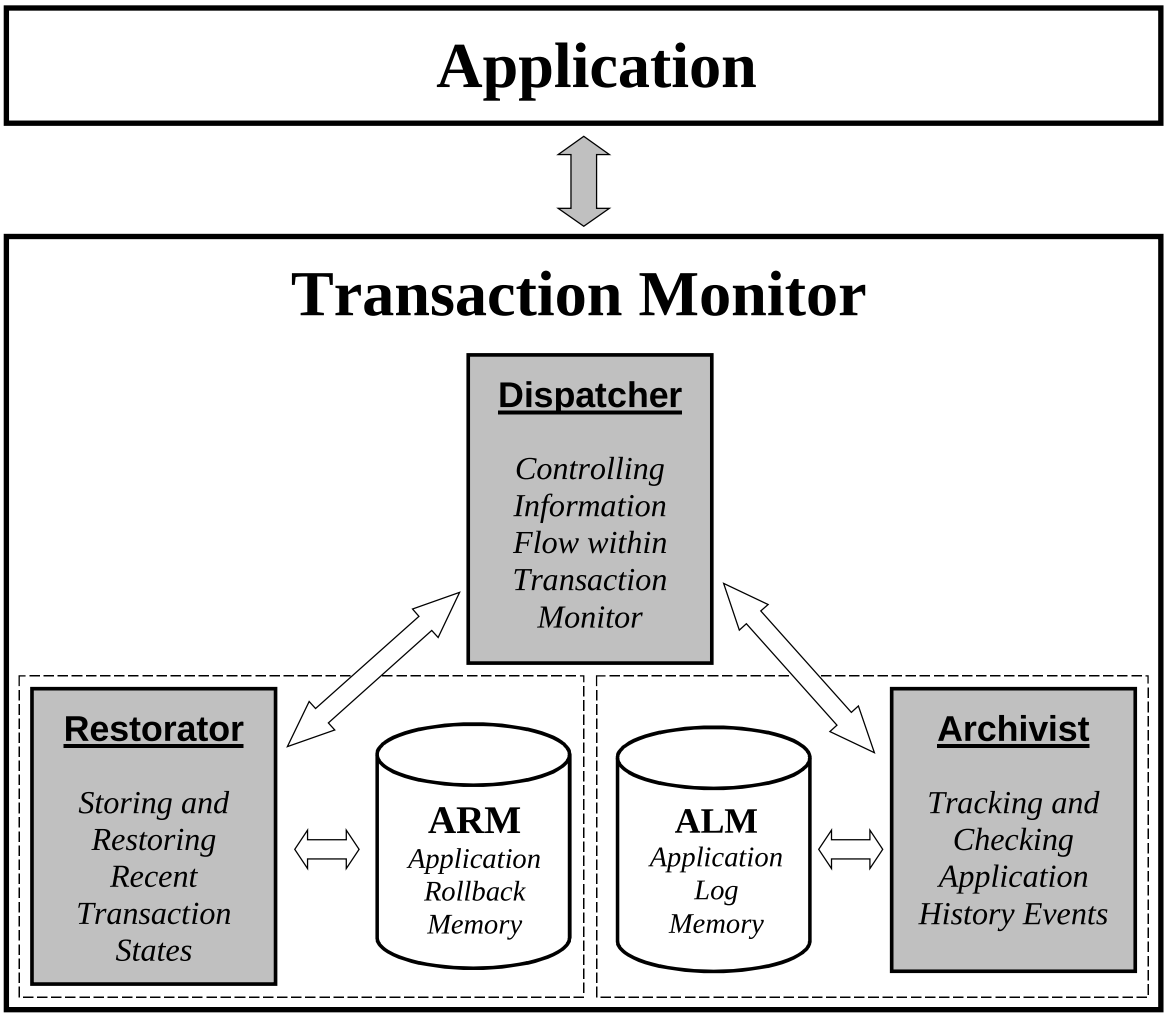

Application
Transaction Monitor
Dispatcher
Controlling Information Flow within Transaction Monitor
Restorator
Storing and Restoring Recent Transaction States
ARM
Application Rollback Memory
ALM
Application Log Memory
Archivist
Tracking and Checking Application History Events

Figure 70. General Architecture of Application-Driven Transaction Monitor

The necessity of the two modules (Archivist and Restorator) and two associated with them memory structures (Application Log Memory and Application Rollback Memory) can be explained by the dual nature of the Transaction Monitor.

On the one hand, the Transaction Monitor should keep tracks of basic Application events to be able to analyze the last state of aborting transaction(s) when the Application is started again after interruption. The Archivist will cover this part of the Transaction Monitor functionality by managing the Application Log Memory.

On the other hand, the Transaction Monitor should store some necessary Application data to be able to continue or rollback the aborted transaction depending on its state. The Restorator will cover this part of the Transaction Monitor functionality by managing the Application Rollback Memory.

In Figure 71 the Transaction Monitor architecture is presented with basic Application queries to it. One can see that some of Application queries will be forwarded only to Restorator, some - only to Archivist, and some - to both of them.

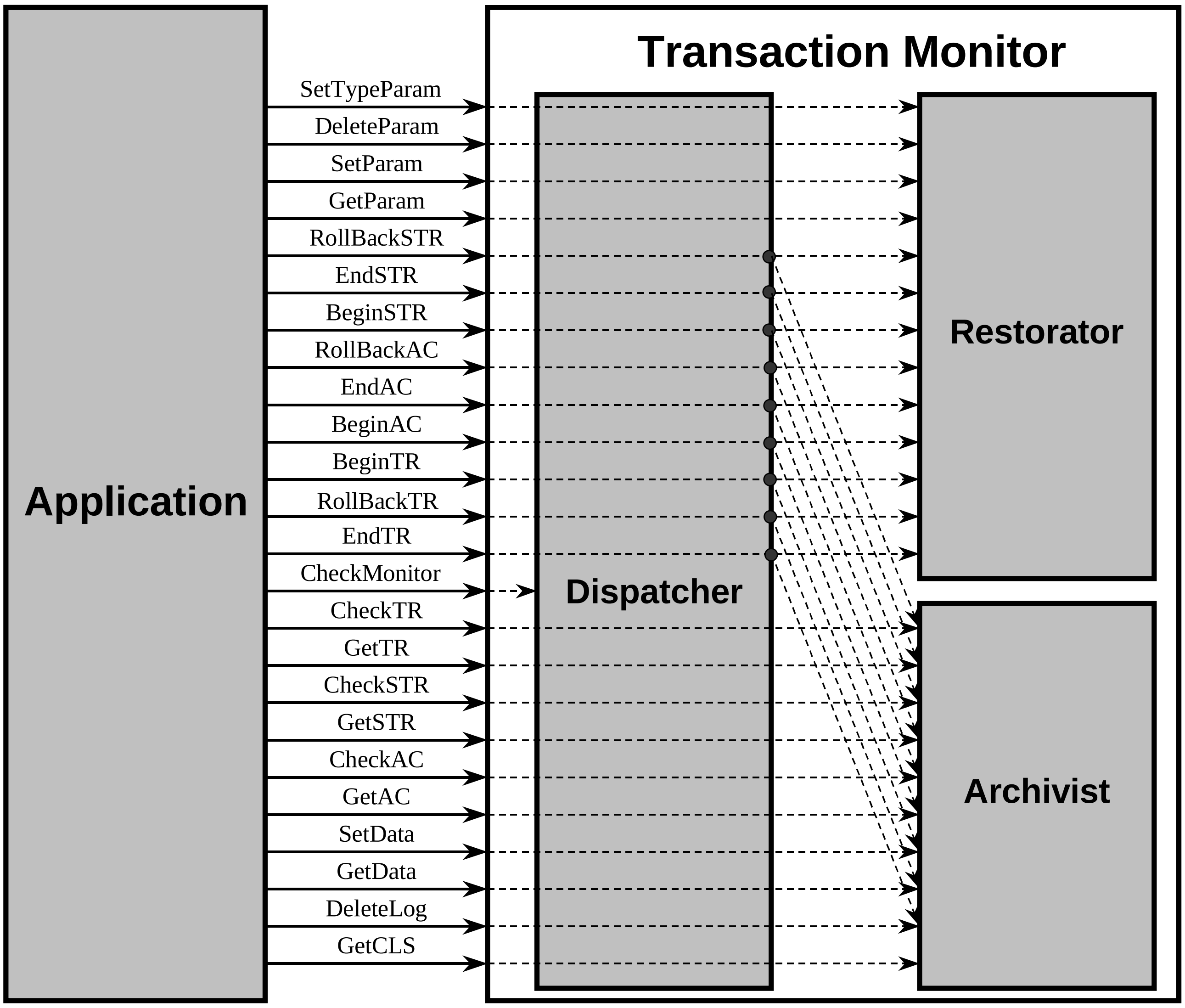


Figure 71. Application - Transaction Monitor interface

9.2 Management of the Transaction Monitor’s memory

Figure 72 illustrates the work of the Restorator as a manager of the Application Rollback Memory (ARM). ARM is a four-layer memory structure, which keeps the values of the Application’s parameters necessary for recovery of mobile transactions in the case of abortion.

Restorator receives parameters online from the Application and keeps their latest values at the Recent Layer. Thus Recent Layer can be used to restart anytime the Application from the point of abortion. Application can set any parameter at the Recent Layer by sending to the Transaction Monitor the SetParam query, define possible behavior of that parameter within the ARM by SetTypeParam query, delete parameter by DeleteParam query, and read parameter when necessary for a recovery by GetParam query.

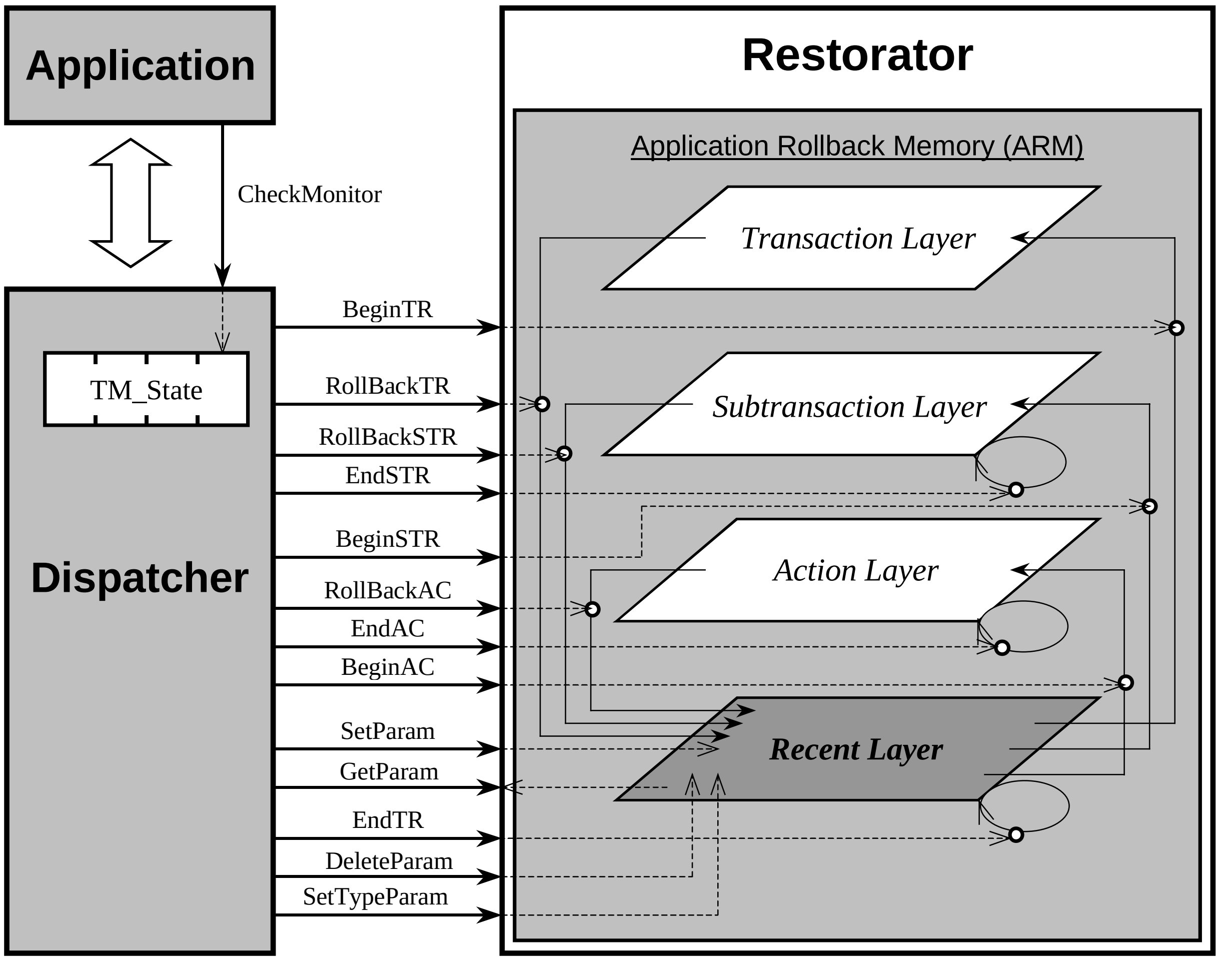


Figure 72. Managing of the Application Rollback Memory by Restorator

Transaction Layer of the ARM is filled after receiving BeginTR query by copying Recent Layer to the Transaction Layer. This makes possible to rollback appropriate transaction from any point of interruption to the very beginning. Such rollback is made by the RollBackTR query, after which the Transaction Layer of the ARM is being copied back to the Recent Layer.

Subtransaction Layer of the ARM is filled after receiving BeginSTR query by copying Recent Layer to the Subtransaction Layer. This makes possible to rollback appropriate subtransaction from any point of interruption to the very beginning. Such rollback is made by the RollBackSTR query, after which the Subtransaction Layer of the ARM is being copied back to the Recent Layer.

Action Layer of the ARM is filled after receiving BeginAC query by copying Recent Layer to the Action Layer. This makes possible to rollback appropriate action from any point of interruption to the very beginning. Such rollback is made by the RollBackAC query, after which the Action Layer of the ARM is being copied back to the Recent Layer.

Cleaning of the Recent Layer, Subtransaction Layer and Action Layer from the data is made in accordance with appropriate queries: EndTR, End STR, and EndAC.

Decision about, which parameter can be moved from layer to layer and which cannot, or which parameter can be cleaned from layers and which cannot, is made by Restorator according to the parameter type assigned by the Application query SetTypeParam.

Figure 73 illustrates the work of the Archivist as a manager of the Application Log Memory (ALM). ALM is a two-layer memory structure, which keeps the record of basic transactional events with appropriate temporal information and necessary clusters of data, which describe information exchange of the Application with another applications during the transaction.

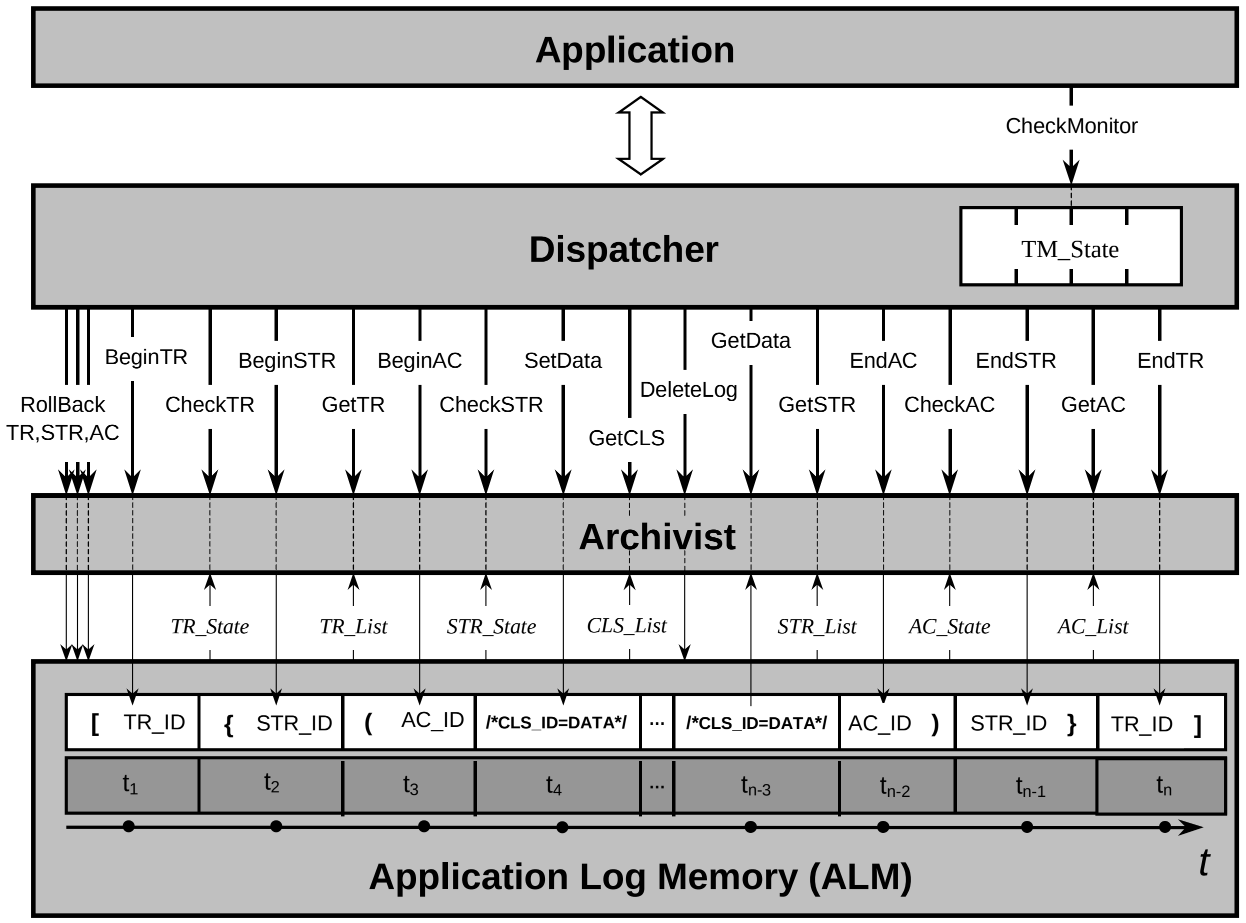


Figure 73. Management of the Application Log Memory by Archivist

Archivist processes queries BeginTR, EndTR, BeginSTR, EndSTR, Begin AC, and EndAC by assigning Ids for appropriate transactions, subtransactions and actions, storing appropriate marginal events with appropriate time values in the ALM.

SetData query results to creation of a data cluster in ALM and recording necessary data from Application there.

Queries GetTR, GetSTR, GetAC, and GetCLS are used by the Application to pick up from the ALM Ids of transactions within temporal interval, Ids of subtransactions within a transaction, Ids of actions within a subtransaction, Ids of data clusters within an action.

Queries CheckTR, CheckSTR, and CheckAC return results of analysis of a transaction, a subtransaction, or an action respectively to allow Application to decide what to do with previously interrupted transaction (subtransaction or action).

Query DeleteLog takes away from the ALM records related to appropriate transaction (subtransaction, action, or data cluster).

Queries RollBackTR, RollBackSTR, and RollBackAC result to cleaning appropriate transaction (subtransaction or action) records from the ALM from their starting point up to interruption point.

9.3 Description of the parameters, queries and modules within the Transaction Monitor

Parameters' types: transactional dynamic (TD) - default type; transactional static (TS); inter-transactional dynamic (ITD); inter-transactional static (ITS).

TD-Parameter can move from layer to layer of the ARM and will be removed from the Recent Layer when the transaction, which stored it, comes to the end.

TS-Parameter, once being stored at the ARM, exists only at its Recent Layer and will be removed when the transaction, which stored it, comes to the end.

ITD-Parameter can move from layer to layer of the ARM and will be left at its Recent Layer when the transaction, which stored it, comes to the end.

ITS-Parameter, once being stored at the ARM, exists only at its Recent Layer and will be left on it when the transaction, which stored it, comes to the end.

Dispatcher receiving any query from the Application fills in appropriate way the Transaction Monitor State (TM_State) to protect atomicity of internal TM transactions:

TM_State: <Query_ID, Query_Flag>,

where Query_ID - Id of the last query received from Application as follows:

$$Query\_ID = \begin{cases} 0, & if \text{ last query } was\ BeginTR; \\ 1, & if \text{ last query } was\ BeginSTR; \\ 2, & if \text{ last query } was\ BeginAC; \\ 3, & if \text{ last query } was\ EndTR; \\ 4, & if \text{ last query } was\ EndSTR; \\ 5, & if \text{ last query } was\ EndAC; \\ 6, & if \text{ last query } was\ RollBackTR; \\ 7, & if \text{ last query } was\ RollBackSTR; \\ 8, & if \text{ last query } was\ RollBackAC; \\ 9, & if \text{ last query } was\ SetTypeParam; \\ 10, & if \text{ last query } was\ SetParam; \\ 11, & if \text{ last query } was\ GetParam; \\ 12, & if \text{ last query } was\ DeleteParam; \\ 13, & if \text{ last query } was\ CheckTR; \\ 14, & if \text{ last query } was\ CheckSTR; \\ 15, & if \text{ last query } was\ CheckAC; \\ 16, & if \text{ last query } was\ GetTR; \\ 17, & if \text{ last query } was\ GetSTR; \\ 18, & if \text{ last query } was\ GetAC; \\ 19, & if \text{ last query } was\ SetData; \\ 20, & if \text{ last query } was\ GetData; \\ 21, & if \text{ last query } was\ GetCLS; \\ 22, & if \text{ last query } was\ DeleteLog. \end{cases}$$

Query_Flag is the indicator of successful end of the last query as follows:

$$Query\_Flag = \begin{cases} 0, \; if \; the \; last \; query \; was \; processed \; OK, \; i.e. \; a \; responce \; has \; been \; received \\ \quad from \; all \; involved \; TM \; modules; \\ 1, \; if \; query \; aborted, \; i.e. \; no \; responce \; received \; only \; from \; Restorator; \\ 2, \; if \; query \; aborted, \; i.e. \; no \; responce \; received \; only \; from \; Archivict; \\ 3, \; if \; query \; aborted, \; i.e. \; no \; responce \; received \; both \; from \; Restorator \; and \; Archivist. \end{cases}$$

Just after the Dispatcher gets any query (other than CheckTM) from the Application, it assigns:

- Query_Flag = 1 (if the query should be forwarded only to Restorator);
- Query_Flag = 2 (if the query should be forwarded only to Archivist);
- Query_Flag = 3 (if the query should be forwarded both to Restorator and Archivist).

After that the Dispatcher records Query_ID, and forwards query to appropriate TM modules: Restorator, Archivist, or both Restorator and Archivist when necessary.

If the query has been forwarded to one of modules, then fact of response reassigns Query_Flag = 0.

If the query has been forwarded to both modules and the first response comes from the Restorator, then immediately the Dispatcher reassigns Query_Flag = 2.

If the query has been forwarded to both modules and the first response comes from the Archivist, then immediately the Dispatcher reassigns Query_Flag = 1.

When in the two last cases the second response comes, then the Dispatcher reassigns Query_Flag = 0.

SetParam (Param_ID, Param_Value, returns 'OK') is a query to Restorator, according to which the parameter Param_ID with value Param_Value will be stored in the Recent Layer of the ARM and will be assigned default type - TD.

SetTypeParam (Param_ID, Param_Type, returns 'OK') is a query to Restorator, according to which the parameter Param_ID will be assigned (or reassigned) Param_Type (0 - TD, 1 - TS, 2 - ITD, or 3 - ITS).

GetParam (Param_ID, returns Param_Value) is a query to Restorator, according to which the value Param_Value of the parameter Param_ID is being read and returned from the Recent Layer of the ARM.

DeleteParam (Param_ID, returns 'OK') is a query to Restorator, according to which the parameter Param_ID is being deleted from the Recent Layer of the ARM.

BeginTR (returns TR_ID) is a query to both Archivist and Restorator.
Archivist produces TR_ID, fixes recent time and puts the record '[TR_ID' with appropriate time value to the ALM memory.
Restorator copies the dynamic parameters (TD and ITD types) of the Recent Layer of ARM to the Transaction Layer, replacing previously stored data from it.

BeginSTR (TR_ID, returns STR_ID) is a query to both Archivist and Restorator.
Archivist produces STR_ID, fixes recent time and puts the record '{STR_ID' with appropriate time value to the ALM memory.
Restorator copies the dynamic parameters (TD and ITD types) of the Recent Layer of ARM to the Subtransaction Layer, replacing previously stored data from it.

BeginAC (TR_ID, STR_ID, returns AC_ID) is a query to both Archivist and Restorator.
Archivist produces AC_ID, fixes recent time and puts the record '(AC_ID' with appropriate time value to the ALM memory.
Restorator copies the dynamic parameters (TD and ITD types) of the Recent Layer of ARM to the Action Layer, replacing previously stored data from it.

EndTR (TR_ID, returns 'OK') is a query to both Archivist and Restorator.
Archivist puts the record 'TR_ID]' with appropriate recent time value to the ALM.
Restorator cleans the Recent Layer of the ARM removing all transactional parameters from it (TD and TS types).

EndSTR (TR_ID, STR_ID, returns 'OK') is a query to both Archivist and Restorator.
Archivist puts the record 'STR_ID}' with appropriate recent time value to the ALM.
Restorator cleans the Subtransaction Layer of the ARM removing all parameters from it.

EndAC (TR_ID, STR_ID, AC_ID, returns 'OK') is a query to both Archivist and Restorator.
Archivist puts the record 'AC_ID)' with appropriate recent time value to the ALM.
Restorator cleans the Action Layer of the ARM removing all parameters from it.

RollBackTR (TR_ID) is a query to both Archivist and Restorator.
Archivist cleans from the ALM all records belonging to appropriate transaction starting from '[TR_ID'.
Restorator copies the Transaction Layer of ARM to the Recent Layer replacing only corresponding parameters from it.

RollBackSTR (TR_ID, STR_ID) is a query to both Archivist and Restorator.
Archivist cleans from the ALM all records belonging to appropriate subtransaction starting from '{STR_ID'.
Restorator copies the Subtransaction Layer of ARM to the Recent Layer replacing only corresponding parameters from it.

RollBackAC (TR_ID, STR_ID, AC_ID) is a query to both Archivist and Restorator.
Archivist cleans from the ALM all records belonging to appropriate action starting from '(AC_ID'.
Restorator copies the Action Layer of ARM to the Recent Layer replacing only corresponding parameters from it.

CheckTR (returns TR_State) is a query to Archivist.
Archivist finds records of the last transaction in the ALM, takes temporal margins of it, checks whether it was successfully finished and returns TR_State:

TR_State: < TR_ID, Begin_TR_Time, End_TR_Time, TR_Flag>,

where TR_ID - Id of the last appeared in ALM transaction, Begin_TR_Time - time of the appearance in ALM of the first '[' record of that transaction, End_TR_Time - time of the appearance in ALM of the last record of that transaction, TR_Flag is the following:

$$TR\_Flag = \begin{cases} 0, \ \textit{if transaction ended, i.e. last record of transaction in ALM} \text{ is } \text{'}]\text{'}, \ \textit{or} \\ \quad \textit{no transaction has been started, i.e. no } \text{'}[\text{'} \textit{ has been found in ALM;} \\ 1, \ \textit{if transaction aborted, i.e. last record of transaction in ALM} \text{ is } \textit{not } \text{'}]\text{'}. \end{cases}$$

CheckSTR (returns STR_State) is a query to Archivist.
Archivist finds records of the last subtransaction within the margins of the last transaction in the ALM, takes temporal margins of it, checks whether it was successfully finished and returns STR_State:

STR_State: <STR_ID, Begin_STR_Time, End_STR_Time, STR_Flag>,

where STR_ID - Id of the last appeared in ALM subtransaction of the last transaction, Begin_STR_Time - time of the appearance in ALM of the first '{' record of that subtransaction, End_STR_Time - time of the appearance in ALM of the last record of that subtransaction, STR_Flag is the following:

$$STR\_Flag = \begin{cases} 0, \ \textit{if subtransaction ended, i.e. last record of subtransaction in ALM} \text{ is } \text{'}\}\text{'}, \ \textit{or} \\ \quad \textit{no subtransaction has been started, i.e. no } \text{'}\{\text{'} \textit{ has been found in ALM within the last} \\ \quad \textit{transaction;} \\ 1, \ \textit{if subtransaction aborted, i.e. last record of the subtransaction in ALM} \text{ is } \textit{not } \text{'}\}\text{'}. \end{cases}$$

CheckAC (returns AC_State) is a query to Archivist.
Archivist finds records of the last action within the margins of the last subtransaction within the margins of the last transaction in the ALM, takes temporal margins of it, checks whether it was successfully finished and returns AC_State:

AC_State: <AC_ID, Begin_AC_Time, End_AC_Time, AC_Flag>,

where AC_ID - Id of the last appeared in ALM action of the last subtransaction of the last transaction, Begin_AC_Time - time of the appearance in ALM of the first '(' record of that action, End_AC_Time - time of the appearance in ALM of the last record of that action, AC_Flag is the following:

$$AC\_Flag = \begin{cases} 0, \textit{ if action ended, i.e. last record of action in ALM } \text{is } ')', \textit{ or no action has been started,} \\ \quad \textit{i.e. no } '(' \textit{ has been found in ALM within the last subtransaction of the last transaction}; \\ 1, \textit{ if action aborted, i.e. last record of the action in ALM } \text{is } \textit{not } ')'. \end{cases}$$

GetTR (After_Time, Before_Time, returns TR_ID_List) is a query to Archivist.
Archivist finds records of all transactions in ALM, which have been started not before After_Time value and not after Before_Time value, creates and returns TR_ID_List from the TR_IDs of the appropriate transactions.

GetSTR (TR_ID, returns STR_ID_List) is a query to Archivist.
Archivist finds records of all subtransactions in ALM, which belong to the transaction TR_ID, creates and returns STR_ID_List from the STR_IDs of the appropriate subtransactions.

GetAC (TR_ID, STR_ID, returns AC_ID_List) is a query to Archivist.
Archivist finds records of all actions in ALM, which belong to the subtransaction STR_ID, which belongs to the transaction TR_ID, creates and returns AC_ID_List from the AC_IDs of the appropriate actions.

SetData (TR_ID, STR_ID, AC_ID, DATA, returns CLS_ID) is a query to Archivist.
Archivist assigns CLS_ID to the cluster, which will keep the given DATA (a set of 'Param_ID = Param_Value' pairs), makes a record of that data cluster in a form /*CLS_ID = DATA*/ to the ALM with appropriate time value.

GetCLS (TR_ID, STR_ID, AC_ID returns CLS_ID_List) is a query to Archivist.
Archivist finds records of all data clusters in ALM, which belong to the action AC_ID, which belongs to the subtransaction STR_ID, which belongs to the transaction TR_ID, creates and returns CLS_ID_List from the CLS_IDs of the appropriate clusters.

GetData (TR_ID, STR_ID, AC_ID, CLS_ID, returns DATA) is a query to Archivist.
Archivist finds the record of the data cluster CLS_ID, which belongs to the action AC_ID, which belongs to the subtransaction STR_ID, which belongs to the transaction TR_ID, takes and returns appropriate DATA record.

DeleteLog (LOG_ID, returns 'OK') is a query to Archivist.
Archivist finds the records in ALM, which belong to the LOG_ID (can be TR_ID, or STR_ID, or AC_ID, or CLS_ID), and deletes these records from the ALM.

CheckMonitor (returns TM_State) is a query to Dispatcher.
Dispatcher returns the record of TM_State to the Application.

10 Conclusion

In this report two approaches to the Transaction Monitor implementation were considered. First one is more general approach and it 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. For that we use the ontology-based framework for transaction management so that the Transaction Monitor 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 Transaction Monitor implementation. Another approach is based on the assumption that some specific terminal-based application is already exists, which supports certain transactions with certain services. In this case the TM is intended to be 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. Both approaches seem to be reasonable to implement to be used in the appropriate context.

References

1. 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.

2. U. Leser, Query Planning in Mediator Based Information Systems, Doctoral Dissertation, Technical University of Berlin, Department of Computer Sciences, 28 June 2000, available in: http://edocs.tu-berlin.de/diss/2000/leser_ulf.pdf.

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. H. Schuldt A. Popovici, H.-J. Schek, Give me all I pay for - Execution Guarantees in Electronic Commerce Payment Processes, Workshop Informatik '99 - Enterprise-wide and Cross-enterprise Workflow Management: Concepts, Systems, Applications, Paderborn, Germany, 6 October 1999, pp. 12-19.

8. J. Suutari, Transparencies on the Architecture of Location-based Services in WAP Environment, MMM-Nokia Meeting, November 2000.

9. K. Tanaka, A Set of Transparencies about the Finnish-Japanese 3G Project, Finpro Office, Tokyo, 2001.

10. Transaction Services: Supporting inter-organisational processes, GigaTS Deliverable 1.1.3, Telematica Institute, the Netherlands, November 1999, available in: http://www.telin.nl/dscgi/ds.py/Get/File-2876/D1.1.3v1_Transaction_Services.pdf.

11. J. Veijalainen, H. Tirri, V. Terziyan, M-Commerce Transactions: Hype or a Real Need, Submitted to VLDB-2001.

12. J. Veijalainen, Transactions in Mobile Electronic Commerce, LNCS, Vol. 1773, Springer, 1999, pp. 208-229.

13. What is E-Speak? Product Information, 2001, Hewlett Packard Company, available in: http://www.e-speak.hp. com/product/overview.shtm.