DIPLOMARBEIT

Shared Virtual Space Distribution Manager - SVSDM - Design and Implementation

Ausgeführt am
Institut für Computersprachen
Abteilung für Programmiersprachen und Übersetzerbau
der Technischen Universität Wien

unter Anleitung von
Ao. Univ. Prof. Dipl.-Ing. Dr. eva Kühn
durch

Richard Mordinyi
Humboldtgasse 10/6/6301
A-1100 Wien
Matr.Nr.: 9825381

Wien, Jänner 2005 ----------

Abstract

Today's mobility requires the capability of having access to resources, like data, any time and anywhere. Actually this is not so difficult. The task is far more complicated however, if you look at it the other way round: How can you easily push data to devices, to workers or to specific groups of people that are both mobile and most of the time offline as well.

This thesis tries to give an answer to this question by offering a pattern called "Shared Virtual Space Distribution Manager". The prototype demonstrates how particular working packages, or more exactly workflows, can be distributed to participating mobile workers by using the space based computing paradigm CORSO.

Kurzfassung

In der heutigen Welt ist es immer wichtiger und immer selbstverständlicher geworden, von überall aus und zu jedem Zeitpunkt auf Daten zugreifen zu können. Diese Problematik der Mobilität ist meiner Ansicht nach schon hinreichend gelöst worden. Mehr Kopfzerbrechen würde aber die Frage verursachen, wie man Daten an mobile Geräte oder an Personen, die mobil und die meiste Zeit auch offline sind, ohne größeren Aufwand verteilt, beziehungsweise sie ausgewählten Personenkreisen zur Verfügung stellt.

Meine Arbeit versucht auf dieses Problem eine Antwort zu geben, indem sie den „Shared Virtual Space Distribution Manager“ als einen Lösungsansatz vorstellt. Der Prototyp zeigt wie Arbeitsaufträge, im Speziellen Workflows, am Besten mit Hilfe der Space Based Computing Technologie an mobile Agenten verteilt werden können.

Danksagung

Mit dieser Arbeit möchte ich mich in erster Linie bei meinen Eltern ganz herzlich bedanken, die während und insbesondere in der Abschlussphase meines Diplomstudiums viel Opferbereitschaft, Geduld und Zuversicht mir gegenüber gezeigt haben, beziehungsweise mir immer mit Rat unterstützend zur Seite gestanden sind.

Vielen Dank an meine Betreuerin Dr. eva Kühn, die den Anstoß zu dieser Arbeit gab. In guter Erinnerung werden mir die ersten spät nächtlichen Codierungen bleiben, mit denen wir den Grundstein des Projekts gelegt haben.

Meine Anerkennung verdient Roland Lutz, der stets mit konstruktiver Kritik, neuen Ideen und einem unermüdlichen Einsatz dafür sorgte, dass diese Arbeit auf keinen Fall die Strasse des Erfolgs verlässt.

Mein besonderer Dank geht auch an Kathrin Janitsch, die mit Ihrer Grazie dieser Arbeit einen besonderen Touch verliehen und mit Ihrer warmen Ausstrahlung mir viel Energie und Motivation gegeben hat.

Abschließend gilt mein Respekt Marcus Mor, einem außergewöhnlichen Freund und Studienkollegen, der mich immer, sowohl privat als auch beruflich, nach besten Kräften unterstützt hat.

Table of Content

I. Introduction

Mobility has become more and more important in today's economy. Big companies acting globally require from their professionals the ability to speak one foreign language at least and to have spent some time abroad. These specifics give firms the unique opportunity to make use of those employees internationally by building the companies' worldwide presences on their employees' already proven knowledge of acting professionally anywhere in the world. In this way global thinking corporations expect to gain advantages compared to their competitors e.g. in being first on a new market.

For whatever reasons modern nomads act worldwide, they cannot abstain using mobile devices. They depend on the machines' help getting online to receive emails, the latest news, their company's tasks, project descriptions or even phone calls at any time anywhere on the planet.

The use of mobile technology is not restricted to global actors. In our surroundings the use of mobile devices plays as well a crucial role directly or indirectly on all of us. A good example of such a use would be in the world of insurance companies employing lots of mobile workers, agents, selling insurance policies to us or evaluating our claims.

This paper is related to the needs of insurance companies and has the intention to demonstrate how the distribution of different orders to the insurance company's agents can be easily performed by using a new kind of technology.

As starting point the current status of the company and its problems will be outlined and a possible solution will briefly be presented.
Part 2 will discuss the different types of technologies that may be used for distributed communication. This section however will mainly focus on CORSO since this platform represents the basis of the SVSDM prototype.
Part 3 will report about the SVSDM prototype itself. It explains what is meant by distribution, what kinds of different CORSO communication objects exist, how they interact and how they are implemented. Chapter 5 of this section shows an example by imitating a real world scenario and explains how the prototype

might be used. The following chapter of this part will describe the advantages of the SVSDM and gives further ideas, hints to improvements and optimizations. The last chapter shows an alternative way briefly how the SVSDM prototype could have been implemented as well.

I.1. Current Situation

The insurance company has a lot of agents set in field services. The orders they receive via mail come from the "Prisma Server", a mail server that provides both an incoming and an outgoing mailbox for each known client.

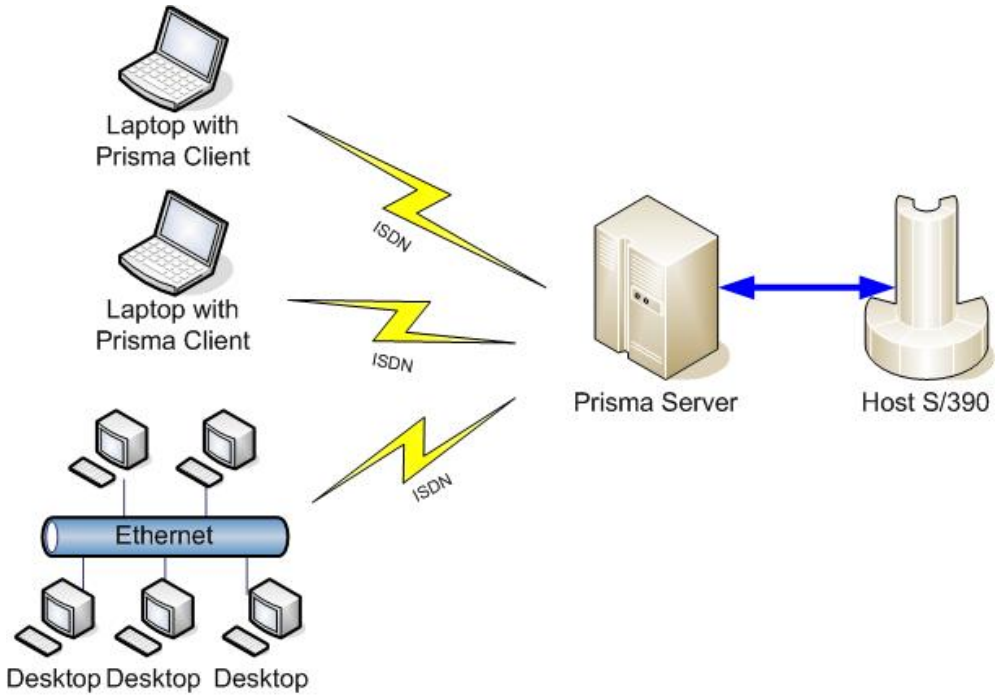


Figure 1: current architecture

Insurance policies are sold via an application running on the desktop computers. Every newly sold policy and any other type of processes having taken place are stored in a database first. After the head of the agency confirmed the stored data they are put into the outgoing mail box and then uploaded to the Prisma Server. There, the incoming data is sent via FTP to the host at predefined time intervals for further processing. Any data coming from the host is placed into the inbox of a specific client. If a client connects to the server its mails are sent to the client where they have to be executed. The results are sent back to the server directly via the in/outgoing mail boxes. Supervisors are not notified about e.g. new contracts at all. They are informed at the end of the month when agents are paid for the contracts they could sell.

I.2. The Goal

This complicated and actually pretty old type of message passing communication system between the different actors in that circle of processing asks for a better solution.
One problem appearing in the described system is the lack of information flow towards the supervisors. Meanwhile the system has been changed, but in a way that still cannot fulfill the requirements of the controllers. Every mail, containing information about finished processes, has to go through the mail box of the supervisor in order to keep her/him informed. This means on the one hand that logging and monitoring of the actions the agents do is possible, but on the other hand this makes the entire system very slow and inefficient.
Another disadvantage that is not possible to solve easily is the way of distribution. Currently an order is dedicated to one agent only. There is no possibility of distribution between the agents. From the supervisor's and from the business' point of view there should be a competition between the agents. This might speed up order processing and bring more satisfied customers. Furthermore the upgrade does not allow publishing specific insurance orders to a specific group of agents either.

The solution would be some kind of a big database that stores every order that has to be executed. This would also require that every agent has the same view of every order. On the basic level no one should be discriminated and no one should be preferred either. All of them should have equal rights in fetching orders. On a higher level however there must be some kind of technology that organizes these orders. It should be able to specify group policies in order to specify, who is allowed to execute an order and who is excluded.
Last but not least the supervisors should have the capability to see what is going on in the database. They should be informed automatically about any action their agents perform at any time. They should also be able to calculate the efficiency of their agents whenever they want.

This thesis will describe how these requirements can be fulfilled easily by using a distribution technology based on a "distributed database", called virtual shared memory.

II. Technical Background

II.1. Types of Communication between Distributed Applications

Usually distributed system applications consist of a number of application components that might be either a process or a software component, for instance an object [4]. These components are often distributed which means that the services they offer run at different computers at different locations.
In order to fulfill the given task both the application and the components among each other require communication facilities. Computer networks and their protocols do not provide the sufficient technology since they do not cover topics like load distribution and balancing or replication of data. For the communication and synchronization of distributed systems there exist two kinds of communication paradigms

- Message-Passing Communication
- Space-Based-Computing

which try to provide some sort of software abstraction above the physical network so that for instance the existence of several independent computers is hidden from the application.

II.1.1. Message Passing

The message-passing paradigm means that components communicate through the explicit sending and receiving of messages. Messages are sent by the client part and received by the server part of the interaction. Messages contain specific information of the task to be carried out by the message receiver.
The development of an application using message passing seems to be easy since the communication protocol is the only barrier that has to be agreed on. But using the Message-Passing paradigm implies both spatial coupling and temporal coupling. The component has to know and explicitly name its communication partner (spatial coupling), and for successful communication both components have to be up and running at the same time (temporal coupling).

An early approach to realize communication in a distributed system based on message passing was to establish a direct communication such as Sockets, RPC or RMI between the components [10, 10]. With this kind of communication each component communicates exactly with those components that can offer the requested services. In the worst case, each component communicates with every other component in the system.

Another approach is to use the client / server model where a few servers provide services to a large number of clients. Clients forward requests to the servers which will send back the response. The appearing problem here is that the server might be a bottleneck if lots of requests are coming to the server. To solve the problem, more servers could be placed in the system which results only in stalling the problem.

Publish / subscribe communication, special form of message passing communication, can also be used in distributed application. A publisher is an application that provides information to the system. The subscriber is the one that uses the information. In this system the publisher does not need to know who uses the information and the subscriber does not need to know who provides the information. In the first place these systems were meant for distribution of information only. Bidirectional communication had to be emulated leading to problems such as scalability.

II.1.2. Space Based Computing

The components of the distributed application use a space for communication. A space is like a distributed shared memory, also called virtual shared memory.

The focus is on the data itself that may be transferred between the components. The notion of a message is not important any more [9]. The shared data objects are now used for the communication. This might also be called the “Blackboard-based communication model” [3].

This leads to some advantages compared to the Message-Passing paradigm, which results in a very flexible system design. First of all the participants do not know anything about each other. This makes it possible to exchange data connectionless and anonymous, since the blackboard is just used to store and retrieve messages. The components do not need to share the same process or machine, but most importantly the participants are temporally independent of one another (temporal uncoupling). Additionally, blackboard models also

provide more security because any execution environment can fully monitor and log all the interactions that occur through its local blackboard.

One of the most popular representatives for space based computing is the "Linda-like Tuple space", e.g. "implemented" in JavaSpaces [5]. It adds important features of the blackboard model by organizing data in tuples and accessing them in an associative way via pattern matching. Since it extends the blackboard model it supports temporal decoupling. By retrieving information in an associative way, Linda-like Tuple spaces support spatial decoupling as well.

CORSO, the basic infrastructure for SVSDM, goes a different way in extending the blackboard model. It can address an object located in the space directly via its Object IDs. This improves scalability and assists at garbage collection. The following chapter will explain CORSO in more detail.

II.2. CORSO

CORSO, standing for Coordinated Shared Objects, is based on research work carried out at the Vienna University of Technology [2]. The idea behind CORSO came up into existence in the late 80s, when GRID [12] became a more important keyword. The first prototype of CORSO was available in 1994 and has been a forerunner for virtual shared memory technologies.

This research focuses mostly on software development in heterogeneous distributed systems that cause complexity and inconvenience for the developer, who has to keep an eye on the following points that should be covered in any good distributed application:

- Location: hide where data is located
- Migration: hide that data might have moved to another location
- Replication: hide that data is replicated
- Access: hide how data is accessed
- Transaction: verified that data is inserted or changed in correct temporal order
- Failure: hide the failure and recovery of data
- Persistence: hide whether data is in memory or on disk

CORSO can be therefore seen as a software layer – middleware – between the distributed application and the operating system that provides a solution for the problems mentioned above in order to disburden the developer. The software abstraction CORSO offers is reached by alleviating the developer with the problems just described and by “merging” the local memory of each client with all the others in the space.

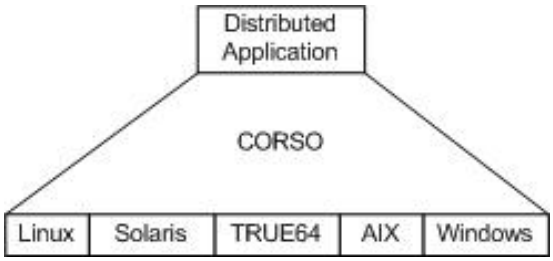


Figure 2: CORSO seen as a Middleware Layer

CORSO runs on the various operating system platforms, as displayed in Figure 2, and represents a connective link for data sharing and synchronization of distributed application in heterogeneous environments.

CORSO poses the basic architecture for SVSDM. In order to know how communication between the participating machines works, it is necessary to explore the way CORSO functions. The following four chapters therefore describe the most important characteristics of CORSO. For detailed information about CORSO please contact [2].

II.2.1. Communication Objects

CORSO uses shared data objects for both communication and for synchronization of parallel and distributed processes. Such objects are uniquely defined by their *OID*s – Object IDentifier. This is a network wide unique identifier of an object. All the replicas of the object are referenced by the same *OID*. How these objects are replicated can be manually defined through the distribution strategy chosen by the programmer.
Such an *OID* is internally built up of three parts. It consists of the IP address of the site where the object was created, of a local counter and of a timestamp. A logical timestamp is assigned to each object that counts how often a value has been written to that object. After successful commitment of the transaction, the counter is increased by one. The timestamp for a newly created object is zero.

CORSO is a tool that helps distributed applications to communicate with each other. Those might also have been written in different programming languages. To overcome this gap CORSO brings its own “language”. Every object that exists in the CORSO space has the *CorsoShareable* interface implemented. This guarantees that for instance the part of the distributed application at site X

written in Java and the one at site Y written in .Net understand each other correctly. *CorsoShareable* can be seen as an interface definition language (IDL) used to represent data written into objects.

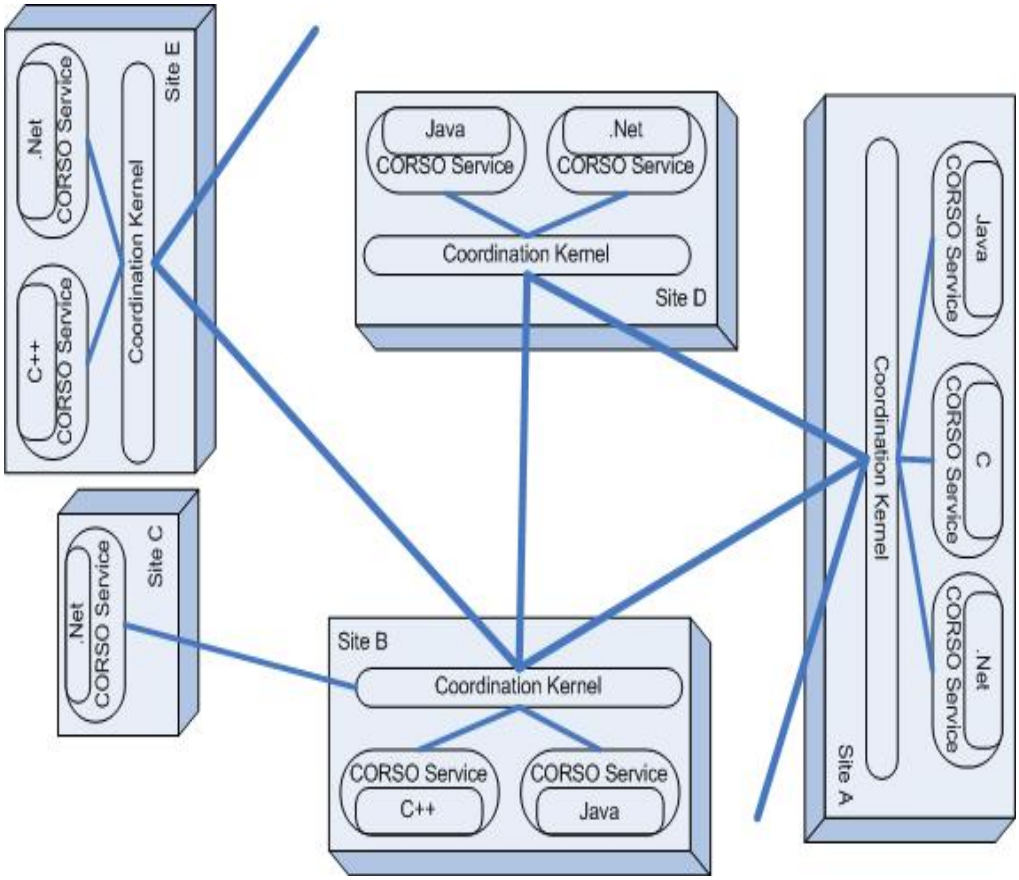


Figure 3: CORSO architecture

CORSO offers two kinds of communication objects through the API specified at creation time.

- Variable Communication Objects

  VARiable communication objects (*VAR*s) behave like ordinary variables in procedural and object-oriented languages. Values can be written into the object as often as necessary.

  The initial logical timestamp is zero increased by one every time a new value has been successfully written into the variable object. The timestamp enables the synchronous read of *VAR* objects and blocks until the next value is written into that object.

For instance, when specifying 2 for the read request the timestamp of the *OID* needs to be at least 3, so that the read request can be performed successfully, otherwise the commitment of the transaction will fail.

- Constant Communication Objects

  CONSTant communication objects (*CONST*s) offer write once behavior. Their state is either undefined or defined. In the first case, their logical timestamp is zero and will be one once defined. In that case a value that cannot be changed again has been written into the object.
  *CONST*s offer some advantages compared to *VAR*s. First of all every site the object has been propagated to can serve local read requests on this object immediately since the value cannot be changed any longer. This implies no handling of transactions and timestamps and offers a better performance compared with *VAR*s. Due to complete independence of other sites sharing the *CONST* object network failures cannot delay requested access to the object.

- Named Objects

  A communication object can be accessed in several ways. Either the reference of the communication object is passed in the list of arguments to a process being started or an object already obtained contains sub objects. In latter case access to the sub object is granted automatically. The third option is to ask the CORSO Kernel for a named object that is registered under a specific name. A named object can be both a constant and a variable communication object.

II.2.2. Notifications

The number of communication objects located in CORSO space can only be limited by the physical limitations of the machine/machines the CORSO Kernel is running on.
In order to stay informed about the most up to date state/value of an object there are two possible ways one can choose:

1. Polling: in this case the value of the object is read at predefined intervals. The method is very inefficient if the value of the communication object hardly changes.

2. Notifications: this is the reason why CORSO supports notification [2, pp. 55]. Here, the CORSO Kernel itself monitors the object and notifies the application if the value of the supervised communication object has changed. This proceeding is very efficient compared to polling.

   Beside that it is fairly easy to use the CORSO API and to take advantage of it. The only thing the programmer has to do, before she/he can start the mechanism, is to create a NotificationItem for the *OID,* the

```
// create Connection to Corso Kernel
conex = modnot.getCorsoConnection();
// initialize notification with all existing notificationItems
CorsoNotification notif = this.conex.createNotification(
                                this.RootWFP.getItems(),
                                new CorsoStrategy(Utils.CorsoStrategyDL_R0));
while (connected && running) {
        CorsoData data = new CorsoData(this.conex);
        // start listening for any change
        CorsoNotificaitionItem fired = notif.start(Utils.TimeOutForNotification, data);
        if (fired != null) {
                WorkFlowPackageContainer struct = new WorkFlowPackageContainer();
                data.getShareable(struct);
                // do something with struct
        }
}
```

Simplified piece of code taken out of tableThreadServer.java demonstrating how to work with CORSO Notifications

   programmer would like to observe, and to add this Item to a CorsoNotification object.

II.2.3. Transactions

In first place transactions in CORSO serve to synchronize object access whether concurrent or not. They can be nested and distributed. Furthermore CORSO supports the parallel execution of sub transactions. Transactions are of prime importance, because the CORSO space can be seen as a big database that

needs to be kept in a consistent way. The important thing is that all running applications have a consistent view of the existing objects they share. CORSO transactions should make this possible.

CORSO provides a nested and powerful transaction model consisting of top level transactions and sub transactions. The first case starts a new transaction scope and is completely independent on any other transactions and no other transaction is dependent on that top level one. On the other hand sub transactions either do belong to a top transaction or are nested in an already existing sub transaction.
All operations concerning variable communication objects are transactional, in the case of *CONST*s the write operation only. In a single transaction, it is made possible to read from and/or write to more than one object. The values that have been changed in an object are going to be visible only if the transaction has committed successfully. The advantage is that this would happen in one atomic step. A transaction containing a read operation can commit successfully only if the object that has been read has not changed its value in the meantime. If it did change then the transaction failed and changes become not visible.

Furthermore CORSO offers two kinds of commitment: “hard” and “soft” commitment. If the transaction has committed successfully both types behave identically. A difference can be only seen if the transaction fails. In case of a hard commit the entire transaction is aborted. In case of a soft commitment the possibility is given to restart the failed sub transaction once again. This saves resources since only the failed part of the transaction has to be redone.

II.2.4. Distribution Strategies

A distribution strategy is some kind of a communication protocol that is used between several distributed CORSO Kernels [2, pp. 117] for communication. It has the responsibility to lead the user to believe in a one big shared memory, based on a distributed architecture.

In CORSO for every communication object a distribution strategy can be assigned to individually. The object receives the necessary information at

creation time. In order to keep up the picture of a Virtual Shared Memory, the distribution strategy has to discuss the following aspects:

- Caching
- Fault-tolerance
- Performance

CORSO's distribution strategy is based on replications techniques. In the current version of CORSO (3.3) there is only one form of that mechanism supported, that would be PR-deep.

PR-deep stands for „Passive Replication with a deep object tree". This form of passive replication is based on a primary copy migration protocol. It is crucial and necessary for the programmer to understand PR-deep, because loss of a primary copy, in which case neither reading nor writing of values is possible, might lead to loss of data. This means that application programs are not independent on the chosen strategy,
In case of a simple program it is not important to pay attention to the distribution strategy. In case of SVSDM it is indeed important to know at any time where all the communication objects are located. To satisfy this need this chapter will explain in the following how CORSO's distribution strategy PR-deep works.

For each communication object existing in the CORSO space, and therefore shared between different sites, an object tree is built up. Every site that has access to that object represents first a node in the tree and second receives administrative information concerning the object. A site can grant access through replication of the object, but it is not necessary that the values stored in the object are replicated as well. This depends on the strategy that is used. Furthermore every node knows its parent and children.

The reason is the following. Every site that wants to have access to a communication object maintains a replica of that object and some more additional information. Every site represents this way a node in the object tree

of the shareable object. The root of the tree is the site that possesses the primary copy. Each other node has only a secondary copy. In order to write a value into that object the protocol demands from the requesting site to own the primary copy first.

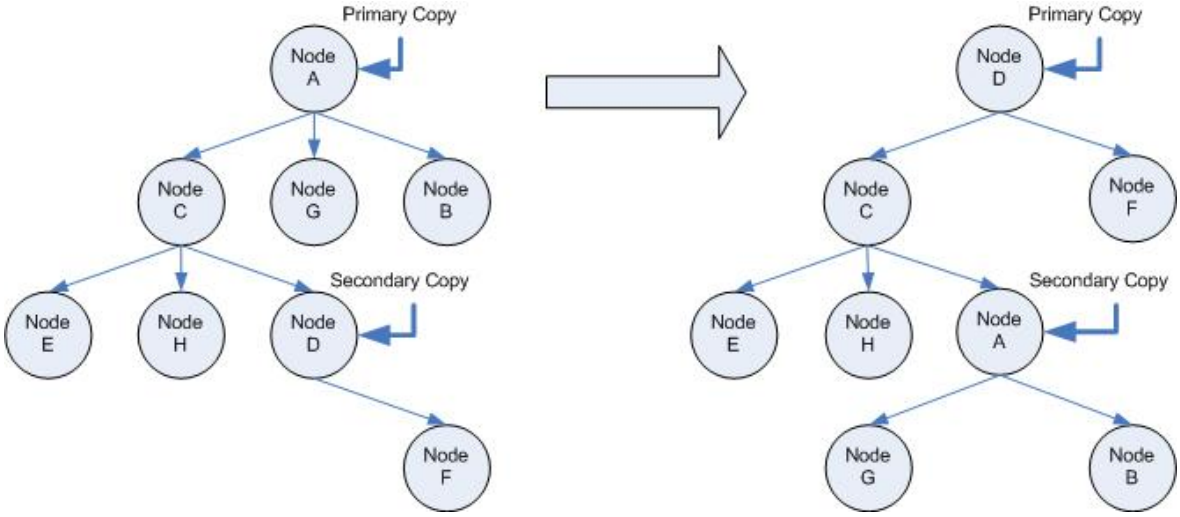


Figure 4: Change of the object tree after movement of the primary copy

When a transaction is about to be committed, the local Kernel tries to fetch the primary copy of the communication object a value has to be written to. If that transaction contains more than one object, the Kernel has to get the primary copies of all of them as well. Only, if all primary copies are possessed by the transaction, it can be committed. These primary copies cannot be transferred to different site's transactions as long as this commitment of the transaction has not been finished.

It is essential to mention that these copies may move between the "starting" and "ending" point of a transaction. This mentioned transaction policy CORSO uses is called Optimistic Concurrency Control. A communication object is unavailable to other users only while the data is actually being updated. The update examines the object and determines whether any changes have been made. This can be achieved by reading the timestamp of the object. Attempting to update an object that has already been changed results in a concurrency violation.

Some of the advantages of PR-deep:

- Network failures are masked
- System failures are masked, if reliability greater than zero is used for communication objects
- Primary copy migration changes the tree by changing the roles but the edges between the nodes stay unaltered.
- Primary copy migration is always a matter between two nodes in the object tree. Whenever the Kernel tries to fetch the primary copy it has to ask its parent for it. Either it has it and can fulfill the request, or it asks its ancestor for the primary copy without bothering the original requesting Kernel (recursive execution). This method prevents broadcasting, because delivery of messages is as local as possible.
- If a connection between two nodes is not available for whatever reasons, the protocol is in the position to be aware of this. In that case it interrupts running transmissions, and restarts them automatically if the broken connection has been reestablished.

The PR-deep protocol supported by CORSO offers via application arguments several possibilities in order to substantiate the chosen distribution strategy:

- Reliability: gives the application the possibility to mask network failures and to store the objects located in the space on a non volatile storage. Actually this is done automatically; the only thing the application has to be concerned with is to specify the strength of the required reliability. In class 0 no data is stored at all. In reliability class 1 communication objects are saved in a data file and CORSO can recover after a site failure. The third option, class 2, is even stronger. It logs actions in a separate file additionally that can be used for recovery, if the data file of class 1 gets damaged. From this point of view class 1 and class 2 reliability can mask possible disk failures as well.
- Lazy / Eager propagation: These two parameters have a big influence on the behavior how communication objects are replicated. Eager means, the

value of the object, including all its sub-objects, are sent to all sites sharing that object, even if it is not sure whether they must see all values. This brings the advantage that the values are available as soon as possible, but it is recommended to use eager propagation only, if the object data must be seen by all sites.

In contrast to eager, lazy propagation distributes the value of the communication object on demand. It is sent to the requesting site only if the data is required there. This is a great benefit and useful if only a few nodes in the object tree are interested in value changes of the object. It makes sense to use this form of propagation if large data, videos or multi-media files, are involved.
Basically the number of messages used by eager or lazy for propagation is quite the same. The important question is when the data is available to be accessed at a particular site.

- Read–Main / Read-Next: These two parameters treat the behavior of how the value of a *VAR* using lazy propagation can be read. The Read-Next method tries to satisfy the request by first looking into the site's local cache to see whether the object value it already has fulfills the read request. This is only possible if the logical timestamp of the object is greater than the specified one. If it is not the case the Kernel tries to contact its parent in the object tree and forwards the request to it. The ancestor will look into its local cache as well. Either, it is able to answer the request and to pass back the value, or it will forward the request to its parent. In the worst case the root node representing the primary copy has to be contacted.

  In the Read-Main method request are always forwarded to the root of the object tree and performed there. This makes sure that the latest value of the communication object is used by the application at read time.

II.2.5. Request / Answer Pattern

Beside the Producer / Consumer Pattern CORSO manual also explains the Request / Answer Pattern (R/A pattern) in order to avoid misuse of CORSO in a client / server oriented programming style. The R/A pattern emulates a Remote Procedure Call (RPC) – server(s) do reply on client requests. This pattern is helpful, if data sharing does not make any sense or whenever an existing RPC application needs to be converted to CORSO.

The R/A pattern is not bounded above to one server. It offers the ability that several servers may response to the requests coming from clients. Figure 5 shows that while *Server 1* handles requests from *Client 1* and *2* Server M responses to requests coming from *Client N* only.

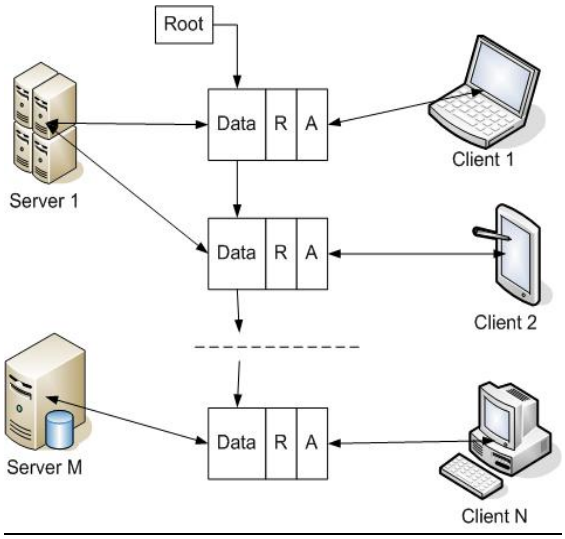


Figure 5: the R/A pattern

As you can see in Figure 5 each client are two variable *OID*s assigned to. They have an "R" and variable "A". Any data written into the first *OID* is a new request coming from a client and to be answered by one of the servers. Data written into the second *OID* is the response to the request that is read by the client and used for further execution.

The ratio of clients to servers is just a matter of required performance and fault tolerance. The more servers the better the performance and the better equipped against server failures, meaning less harm to the entire system. Furthermore it is not necessary to store those variables in a linked list. The linked list data-structure in Figure 5 is just one example how these objects can be manage. (cp Figure 11 and Figure 14).

III.SVSDM

III.1. Introduction into the SVSDM Distribution Theory

III.1.1. The Cycle of Package Distribution

"*A good distributed system should easily connect users to resources; it should hide the fact that resources are distributed across the network; it should be open; it should be scalable.*" [1, pp.4] All these points are sufficiently covered by CORSO. Since distributed applications are more or less dependent on the features the middleware layer underneath offers, they could and also should provide those special characteristics towards the user of the application.
An application that is predetermined for distribution of all kind of objects should be easy and simple to handle. It should be as transparent as possible meaning that the users of the system should only have to know who to supply the system respectively how to grab a package. Users on the one hand are the ones who want to distribute packages and on the other hand the ones who fetch them.

This leads to the "principle of the two ears". On the left "ear" of the circle [Figure 6] the master user just pushes new objects into the space meant for distribution. On the right "ear" independent users spread all over the network only have to choose the desired package and start the application contained in it.

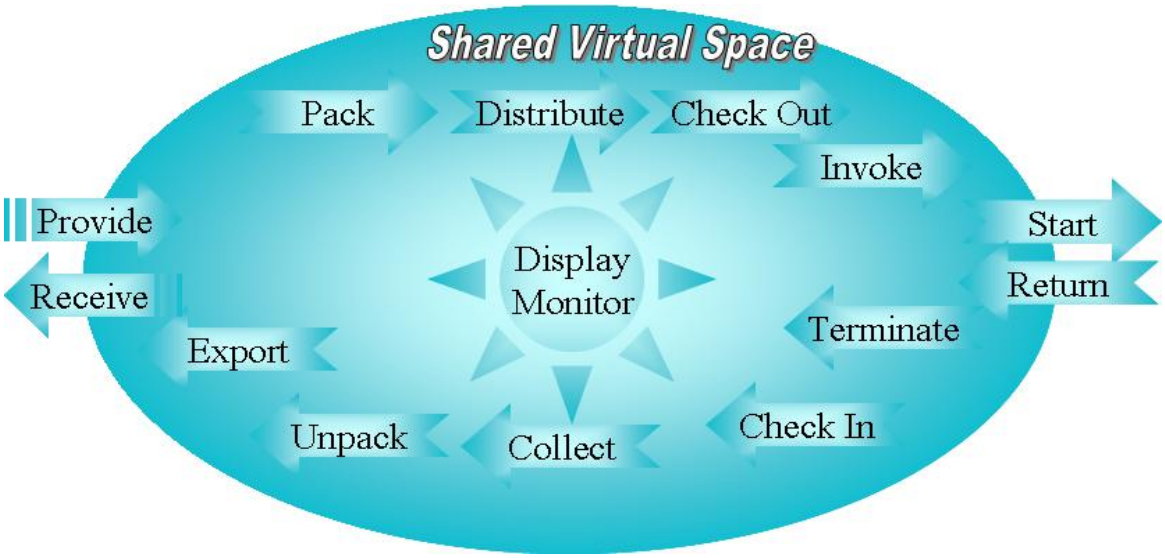


Figure 6: Cycle of Package Distribution

In the following the three main points of the circle are picked out and described in more detail to get a first idea how the SVSDM prototype works. These would be the importation, the distribution and the collection of packages. The other labels of Figure 6 intercept with one of the key points.

III.1.2. Importing a package

In order to be able to distribute anything the system needs data. This information is provided by the user of the system. This could be done either through a GUI or via files. The SVSDM prototype works with files. The user has to specify which files to distribute. This offers on the one hand independence from other systems and on the other hand processes can work automatically with SVSDM through its offered interfaces.

The given data is zipped first and put into a communication object. Another communication object, the package, is created and tagged with additional information necessary to specify the characteristics of the package itself.

The task of the distribution manager is now to put this package into the virtual space. It should be done in a way, so that any other user connected to the space is informed about any updates as soon as possible.

III.1.3. Distribution of a Package

Once the user decides to open a package for distribution, the system copies the package into a special list existing in the space. This list offered by CORSO is the notification service. Every time the user puts a package into that list, CORSO informs every other host about the new package automatically and replicates it to those sites.

At that time all available packages are visible at the worker's site. Actually that should not be the case. The problem is that in the current version of SVSDM the usage of profiles is not fully implemented. As mentioned above a package contains additional information. A part of this information should be used to specify the profile of a package. The user should be able to determine what kind of groups or single users are allowed to see that package in the global space. This means that some of the users do not even see a newly added package. This

method does not only distribute packages within the system, it also distributes them to authorized users as well.

III.1.4. Invoke a package

At this point most of the workers should be informed about the newly added package. Depending on the network, distribution should not take more than a few seconds. The presented information of the available packages shown on an information board should help the user to select the most accurate piece of work.

If the user has found the desired package of choice she/he has to ask the SVSDM to "move" it to the user's local space. This is necessary since offline working modus should be supported as well. Once SVSDM replicated the package and all of its content to the user's site, the global space is updated. This means that on the one hand the package becomes invisible for any other user and on the other hand the user who created the package is informed about the user who selected it.

III.1.5. Execution of the content of a package

After the work has been replicated to the user's site, she/he should have the opportunity to go offline as well. From now on it is up to the application what happens to the content of the package.
For a foretaste, the chosen example uses a *BPEL* engine. It is needed in order to run the content of the package on the client. This consists of workflows and some more Java files.

III.1.6. Collecting a package

In some cases it is not enough to just distribute a package. A number of situations require that the output the application produces is sent back to the global space. This means that there should be the possibility given to perform this task, both manually and automatically via given interfaces.
Once a package has been uploaded, it cannot be selected once again. It is declared as done and removed from the global blackboard. The answer is then stored in the local space of the initiator.

III.1.7. Exporting a package

The initiator can choose between two possibilities. Either he deletes, removes the package from the local space or saves the response. In the first case any answer coming from the worker process is ignored. In case of an export the received zipped data is saved to a user specified directory. After the data has been stored, the user still has the possibility to remove the package. Once a package has been removed from the space it is lost forever.

Summed up the SVSDM prototype has the following responsibilities to fulfill:

- Import and transform provided information into an SVSDM compliant package
- Publish the provided package in the globally shared space
- Visualize imported work packages of the global working list
- Provide the ability to select desired work packages and to move them to the local space
- Visualize selected work packages in the local client work list
- Help to execute the content of the package
- Return result to local client working environment
- Update work package
- Visualize updates in local client work list
- Check in work package updates back to global space
- Visualize updates in global space
- Unpack work package
- Export work package to receiving system

All these actions should happen in a way that leaves the entire "distributed database" consistent. This task can be achieved with help of CORSO's transaction service.

III.1.8. Exception Handling

There are two kinds of ways why an exception can occur. Either the problem is related to the network or an error message appears because of a user error.
Basically, CORSO is able to mask network problems. If CORSO for instance tries to get a primary copy of a communication object, it has to exchange messages with other peers. If the network connection breaks during the communication and has been reestablished within a predefined amount of time, then the application is not notified about anything and can keep on working with the primary copy. However, if the time interval is elapsed, the SVSDM informs the application by throwing a *TimeoutException* due to unreliable network connection.
The second difficulty is concerned with the question when packages are allowed to be fetched and executed by the users. The following points represent rules a user of the SVSDM prototype has to know about:

1. A package cannot be fetched twice at the same time. If two or more users try to get a package all of them, except the first, will receive an error message. The first come first served principle is used.

2. The same package cannot be obtained by the same user at the same time.

3. Over a longer period of time it is possible to select a package twice or even more often. In this case the answer of the one who fetched the package last is valid. Any other responses are ignored.

4. A user is not authorized to receive packages. In that case she/he will receive a message requesting to ask for authorization first.

5. A package can not be fetched if it is not marked *Selectable*.

6. If the user responses to the content of a package that is not available any more, she/he will receive an error message.

An explanation of how the compliance of these rules is observed is given in ch. III.2.1.

As mentioned in the beginning of this chapter the SVSDM should be implemented and work in a way that requires only a minimum of effort on behalf of the user.

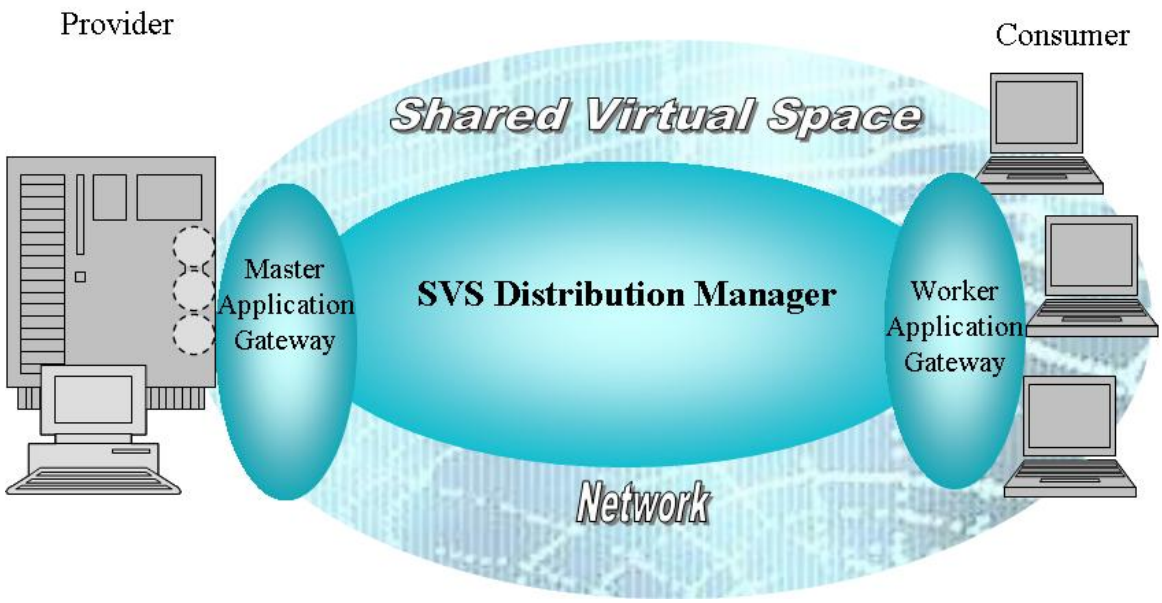


Figure 7: the SVSDM prototype

Figure 7 presents the SVSDM prototype in a way how it should be seen at a global point of view. The Application Gateways on the left and the right site represent the "ears" and provide access to the core SVSDM via its interfaces. Distribution of packages is done via the Distribution Manager placed between them. Behind all these there is CORSO situated, in order to make sure that each action started is performed transactional and without bothering any process in case of failures.

III.2. Architecture

The basic architecture of SVSDM is going to be presented in the following chapter. The reader should get a detailed understanding of how SVSDM functions internally.

In this section it should be clarified how objects are stored, what kind of objects do exist and how those communicate with each other. In the following *package* is used as a generic term for a communication object that stores the files provided by the user and meant for distribution via referencing object *ID*s. The prototype mainly works with workflows and any other files related to it.

Figure 8 shows the entire picture that is taken for the coming explanation of the SVSDM architecture.

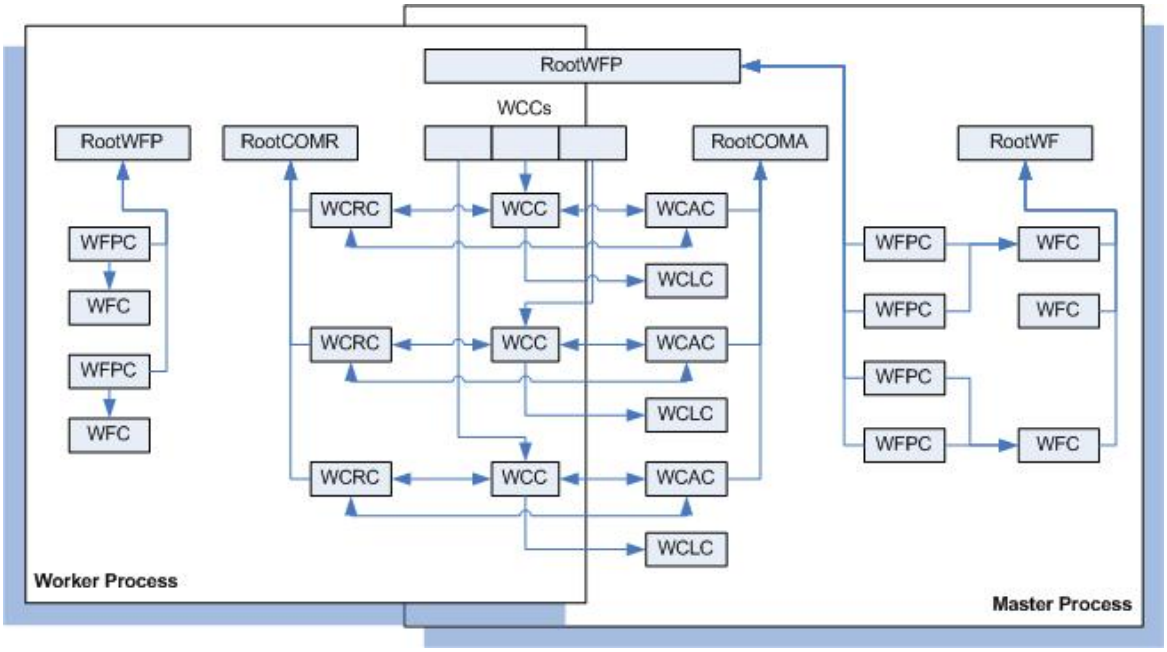


Figure 8: main architecture of SVSDM

III.2.1. Design of Classes

According to Figure 8 there are two kinds of processes that may gain access to the SVSDM communication objects. The master process is the one that inserts packages into the space and the worker process the one that fetches them and performs their content. This simple explanation for this chapter is adequate for the first, but more detailed information about the two processes can be found in chapter III.2.3.

A communication object, that needs to be stored and made available in the CORSO space, has to implement *CorsoSharebale*. Beside the Notification objects offered by CORSO that have this interface implemented automatically there are two objects of main importance in SVSDM. The first one is the *WorkFlowContainer* and the second the *WorkFlowPackageContainer* which represent data objects that are exchanged between the different processes. Additionally there are also the *WorkerCommunicationRequestContainer*, *WorkerCommunicationAnswerContainer*, *WorkerCommunicationContainer*, *WorkerCommunicationLogContainer* and *WorkerCommunicationContainers* that have the interface implemented in order to make communication between the running processes possible.
These seven classes, necessary to run the SVSDM prototype, are explained in the following.

- WorkFlowContainer (*WFC*)

  Actually this object is not a part of the Distribution Manager. It stores

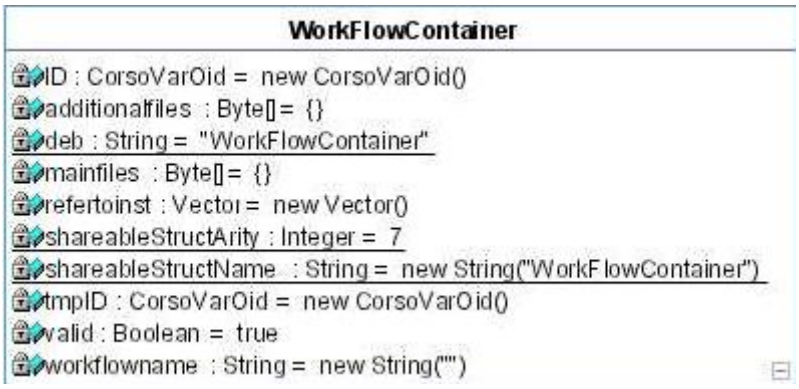


Figure 9: WFC

  all the information that is relevant for a workflow to be executable at worker's site. A workflow, that describes business processes, consists mainly of *WDSL* and *BPEL* files [6, 6]. The Web Service Description Language (*WSDL*) outlines the internal structure of the sent message, how it has to look like in order to be understood by any other Web Service. The Business Process Execution Language (*BPEL*) on the other hand describes the workflow itself. It is a sequence of different commands, like if and while, that try to map a business process in a way understood and correctly executable by a computer. In order to fulfill its task and to be able to coordinate itself a *BPEL4WS* process can send to and receive

messages from other Web Services. Each of these messages can have its own *WSDL* file as well. So, at the end a workflow consists of a *BPEL* and at least of one *WSDL* file. Since it is a little bit difficult to distinguish between the different *WSDL* files, it was easier to define two variables. The *BPEL* file and the main *WSDL* file are stored in *mainfiles*. Any other *WSDL* files used by the workflow are saved in *additionalfiles*. In the latter, Java files can be saved as well. They are necessary if the workflow should be supported with a GUI at worker's site. Additional jar files to be able to run the Java program are also permitted.

Variables of the object:

- ID: stores the object's *OID* at time of creation
- tmpID: stores the *OID* of the object's clone at worker site
- valid: indicates whether the object is still used or meant for deletion
- workflowname: stores the name of the workflow
- mainfiles: contains the *BPEL* and the main *WSDL* files
- additionalfiles: includes any additional files, like more *WSDL* or Java files
- refertoinst: stores the name of the *WorkFlowPackageContainer* referencing to this object

- WorkFlowPackageContainer (*WFPC*)

This object is the one that is used by the Distribution Manager to publish all available packages. This is the one that is replicated to all worker processes. The *WFPC* can be seen as the envelope of a letter with some additional information needed for distribution instead of the addressee. The letter itself, the Workflow, is referenced through an *OID*.

Furthermore, this object has the responsibility to offer monitoring possibilities. This is necessary in order to avoid any misuse. Let us assume that a worker has fetched an object, goes offline and for whatever reasons never online again. In this case the package would remain selected forever and never performed.

Figure 10: WFPC

Therefore, the class allows defining for how long a package can be seen in the pool of packages or how long it can remain at worker's site. These are necessary in order to give the master process the ability to define deadlines. The first case is optional – either the package is in the space for a predefined amount of time or until it is deleted. The second case can be seen as some kind of protection. The reason for this mechanism is that there might be the possibility that a worker fetches a package and then disappears with it for some while. The master process takes care of these circumstances and put things straight by reallocating the package, if the defined value has been exceeded. This means that now any other worker process can grab the package again and work with it. This implies that in the worst case the same workflow has been performed as often as often it has been fetched.

In most of the cases workers are very mobile and not tied to a specific working area. The workers may travel between several cities to fulfill their tasks. Insurance agents for instance would need some area specific data of their customers in order to improve their sales talks. This data can be saved in *instancefiles*. The workflow they require for those files are referenced by *workflowref* – the letter itself.

Variables of the object:

- ID: stores the object's *OID* at time of creation
- tmpID: stores the *OID* of the object's clone at worker site
- valid: indicates whether the object is still used or meant for deletion
- begindatequeue: stores the information when the object was created and placed into the space
- ttlcontainer: stores the amount of time the object is allowed to remain in the space
- begindateworker: stores the date when the object was selected by a worker. The variable is set every time the package is fetched
- ttlworker: stores the amount of time the object is allowed to remain as selected
- selected: indicates whether the package has been selected or not
- selectedby: the name of the worker who fetched the package. This worker may also response to the content of the package
- status: stores the status of the package. This might be either *Selected*, *Selectable*, *Reselectable*, *Retransferred* or *Expired*.
- instancename: name of the object
- instancefiles: stores additional data files needed to be connected to the package
- workflowref: contains the *OID* of the referenced *WorkFlowContainer*
- workflowname: stores the name of the referenced *WorkFlowContainer*
- answered: indicates whether there is an answer or not
- answertitle: stores the title of the response
- answerfiles: obtains the response itself

- WorkerCommunicationContainers (*WCCs*)

  According to the R/A pattern it is necessary to provide a communication object in order to make communication between the master and the worker process possible. Since there can be more than one worker connected to the space several entities of this object are required.

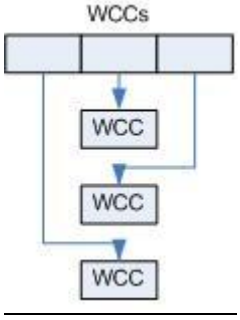


Figure 11: WCCs

For this reason another object (*WCCs*) is needed that has the only task to store all the references to these communication objects.

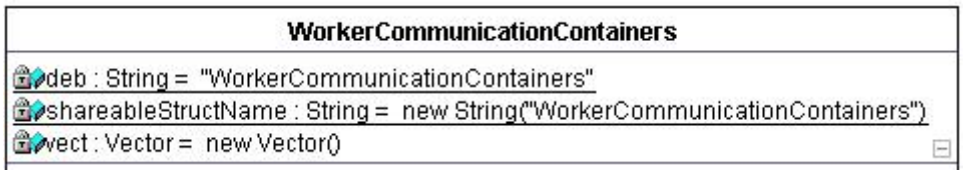


Figure 12: WCCs

Variables of the object:

- vect: stores the *OID* of each *WorkerCommunicationContainer*

- WorkerCommunicationContainer (*WCC*)

This object keeps the references to worker specific communication objects. Each worker that wants to talk with the master process receives, according to the R/A pattern, at least an answer and a request communication object. Since SVSDM offers logging services additionally, every *WCC* stores the *OID* of another CORSO object (*WCLC*) that provides the facilities for that service.

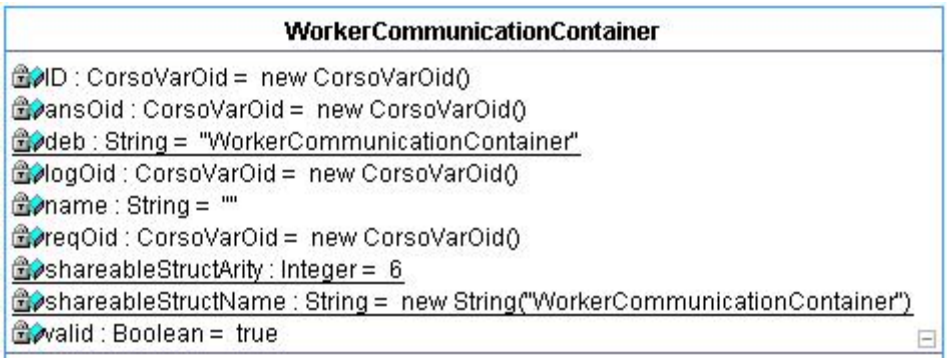


Figure 13: WCC

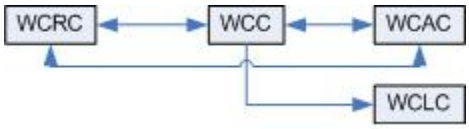


Figure 14: WCC and its related objects

Variables of the object:

- ID: stores the object's *OID* at time of creation
- valid: indicates whether the object is still used or meant for deletion
- logOID: stores the *OID* of the object that is used for logging
- ansOID: stores the *OID* of the object that is used to place responses on requests
- reqOID: stores the *OID* of the object that is used to place requests coming from the worker process
- name: stores the name of the only user who is allowed to use the referenced objects

- WorkerCommunicationRequestContainer (*WCRC*)

This communication object and its converse partner, the AnswerContainer, cannot function usefully without the notification service provided by CORSO. Both of the objects are completely used for communication between the worker and master processes. This exchange of messages is used for e.g. synchronization (logging) or for sending of requests (fetch packages). For a detailed explanation of how the two processes communicate with each other see chapter "III.2.4 Communication between Master and Worker Spaces". There is also explained how the notification service is used and why synchronous reading is not sufficient.

The *WCRC* object is used by the worker process only. It places a request that is read by the master process. The message ID (*MSGID*) identifies the intention of the worker and according to this ID the master process executes different actions.

```
WorkerCommunicationRequestContainer
ID : CorsoVarOid = new CorsoVarOid()
MSGID_ANSWER : Integer = 0
MSGID_ANSWER_LENGTH : Integer = 1
MSGID_DELETE : Integer = 5
MSGID_DELETE_LENGTH : Integer = 0
MSGID_HIGHEST : Integer = 5
MSGID_LOG : Integer = 1
MSGID_LOG_LENGTH : Integer = 2
MSGID_RELEASE : Integer = 3
MSGID_RELEASE_LENGTH : Integer = 1
MSGID_SELECT : Integer = 2
MSGID_SELECT_LENGTH : Integer = 2
MSGID_SETANSWER : Integer = 4
MSGID_SETANSWER_LENGTH : Integer = 3
POS_ANSWER : String = "ACK"
ansOid : CorsoVarOid = new CorsoVarOid()
data : Vector = new Vector()
deb : String = "WorkerCommunicationRequestContainer"
msgid : Integer = -1
parent : CorsoVarOid = new CorsoVarOid()
reqNr : LongInteger = (long)-1.0
shareableStructArity : Integer = 5
shareableStructName : String = new String("WorkerCommunicationRequestContainer")
```

Figure 15: WCRC

Each request is marked with a random ID as well, in order to make each started communication unique and to know if the received answer is the appropriate one.

Variables of the object:

- ID: stores the object's *OID* at time of creation
- parent: stores the *OID* of the appendant *WCC*
- ansOID: stores the *OID* of the object that is used to place the response on this request
- msgid: the request itself indicating what the worker process would like to see performed. The ID of the message can be chosen from one of the MSGID_ variables only.
- reqnr: a random number to make to request unique
- data: any data needed to perform the request correctly

- WorkerCommunicationAnswerContainer (*WCAC*)

This object is used by the master process only. After a request has been placed by the worker process and the master process notified by CORSO a new thread is started where the request is performed. The result is then

placed into the worker's answer object. On the other hand this will inform the worker to continue the work.

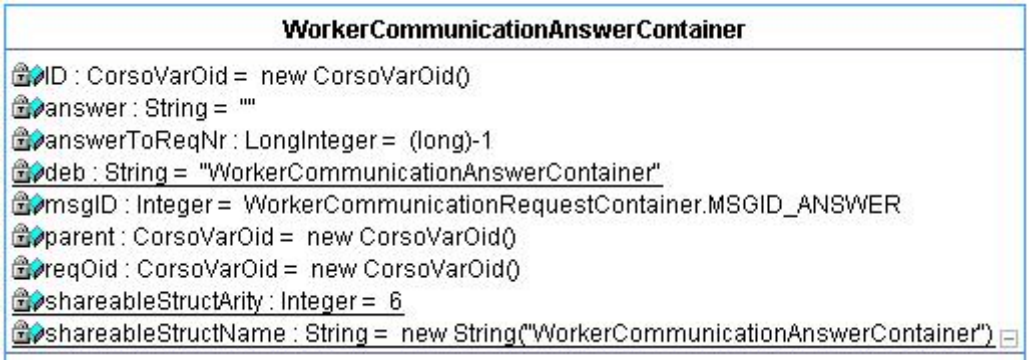


Figure 16: WCAC

The communication ID of the result is exactly the same received with the request. The answer to the request is either according to the message ID an acknowledgment or a request made by the master process. Any other types of answer are interpreted as an error message and handled that way.

Variables of the object:

- ID: stores the object's *OID* at time of creation
- parent: stores the *OID* of the appendant *WCC*
- reqOID: stores the *OID* of the object where the request has been placed
- msgid: the ID of the message is always an acknowledgement
- answerToReqNr: the *reqnr* number appearing in the request
- answer: the answer to the request that might be either a request, an acknowledgement

- WorkerCommunicationLogContainer (*WCLC*)

This object stores all the activities a worker does. It logs how often a package has been fetched or how often a package has been redelivered. In the latter case there are some distinctions to make. It is necessary to know if the package has been redelivered on time or too late. Does it contain any information or is it empty and so has not even been touched by the worker? These pieces of information help the user of the master process to get a picture of its agents and to evaluate them. The main

problem that appeared here was not how to collect data but how to synchronize these between the worker and master processes. The important question was which timestamp to take in case an event occurs. The date at the master process cannot be taken because this information could not be fetched if the worker is offline. A mixture of master time and worker time is neither meaningful since synchronization of them would cause a lot of work.

The solution that has been chosen is that every event that has to be logged is marked with the worker's timestamp. This time will also be used at the master process without any changes. To keep the master process up to date, the logged data there is updated by the worker process. Updates do occur every time whenever the worker process sends a request to the master process. In the background updates of the collected data are sent to it as well.

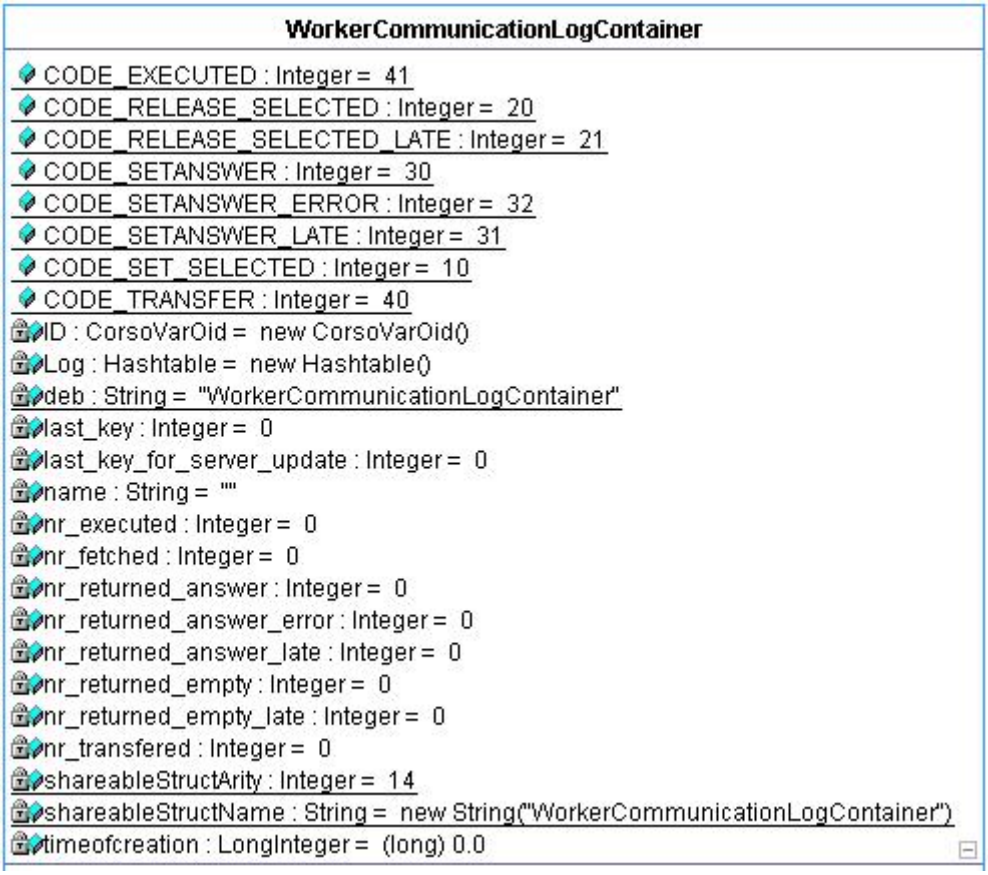


Figure 17: WCLC

Variables of the object:

- ID: stores the object's *OID* at time of creation
- name: stores the name of the worker

- timeofcreation: indicates when the object was created
- Log: stores *"code_when_what"* – entries. *Code* is one of the CODE_* variables saying what kind of action took place. *When* refers to the date the action happened while *what* represents the log message. Logging itself is performed via the private method *appendToLog* .

```
private boolean appendToLog(int code, String time, String logmsg) {
  String newlog = new Integer(code).toString()+"_"+time+"_"+logmsg;
  // analyse code and update appropriate global variable
  switch (code) {
    case WorkerCommunicationLogContainer.CODE_SET_SELECTED:
      this.nr_fetched++; break;
    case WorkerCommunicationLogContainer.CODE_RELEASE_SELECTED:
      this.nr_returned_empty++; break;
    case WorkerCommunicationLogContainer.CODE_RELEASE_SELECTED_LATE:
      this.nr_returned_empty_late++; break;
    case WorkerCommunicationLogContainer.CODE_SETANSWER:
      this.nr_returned_answer++; break;
    case WorkerCommunicationLogContainer.CODE_SETANSWER_LATE:
      this.nr_returned_answer_late++; break;
    case WorkerCommunicationLogContainer.CODE_SETANSWER_ERROR:
      this.nr_returned_answer_error++; break;
    case WorkerCommunicationLogContainer.CODE_TRANSFER:
      this.nr_transfered++; break;
    case WorkerCommunicationLogContainer.CODE_EXECUTED:
      this.nr_executed++; break;
    default:
       // wrong code is used – entry cannot be logged
      return false;
  }
  last_key++;
  this.Log.put(new Integer(last_key),newlog);
  return true;
 }
```

It takes the given *code* argument and analysis it. Depending on the number the appropriate global variable is incremented. Those variables are used to avoid difficult and time consuming analysis of the entire log table. These variables allow getting any provided information immediately.

- last_key: the highest key in the hash table
- last_key_for_server_update: stores the key that was used last for an update at the site of the master process. Every entry coming after this key is not mirrored there. In case of an update *last_key* equals *last_key_for_server_update*
- nr_fetched: stores the number of fetched packages
- nr_returned_empty: logs the number of packages that have not been answered
- nr_returned_empty_late: stores the number of packages that have not been answered and "sent" back to space after the deadline
- nr_returned_answer: represents the number of packages that have been answered
- nr_returned_answer_late: represents the number of packages that have been answered after the deadline
- nr_returned_answer_error: represents the number of packages that have been answered after the deadline but had to be refused. This is the case if the package has been selected or even already answered in the meantime by another worker.
- nr_transfered: stores how often workflows have been deployed
- nr_executed: logs how often the provided Java applications have been started

III.2.2. Data Structures

In the first chapter it could be read that the shared objects, required for communication between the worker and master processes, are pooled in one object - *WCCs*. This vector of *OID*s is used to find the worker specific *WCC* object whenever it creates a connection to the master space after it has been offline.

This search method is used at the same time to check whether the worker process is authorized to interact with the master process. If the worker's *WCC* is contained in the vector then the agent is authorized otherwise it will be refused. This means that such a *WCC* and its appending objects are created only whenever the user of the master process authorizes an agent.

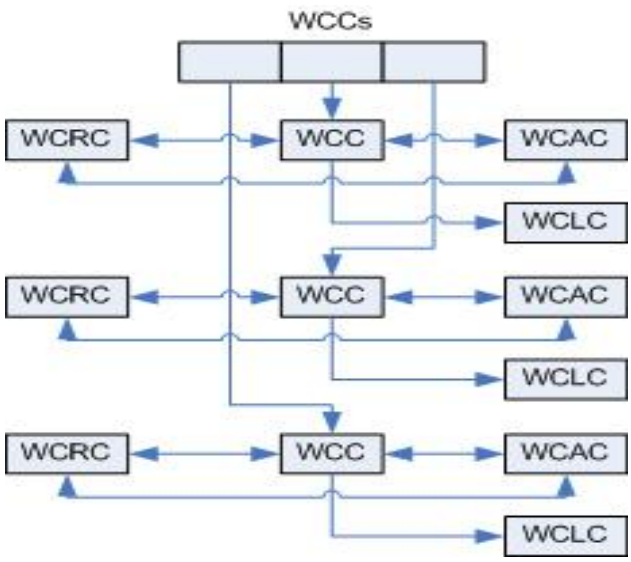


Figure 18: objects used for communication

The objects, *WCRC* and *WCAC*, are also stored in a notification list. This is necessary otherwise there would be no interaction between worker and master process possible. The reason for this chosen structure is explained in the next chapter.

A reliable, meaning at least reliability class 1, notification vector offered by CORSO is used to store the packages SVSDM distributes. Actually there are two of them. The first one is *RootWFP* and the second *RootWF*. This would not be necessary but the advantages such a CORSO service offers cannot be swept aside. First, if a vector like *WCCs* was used for storing, then every process, that wanted to know all available packages, had to create an unreliable notification for its own. This would have cost time and wasted resources. With the current solution the processes fetch the main notification object and at the same time they will receive all offered services as well.

III.2.3. Processes

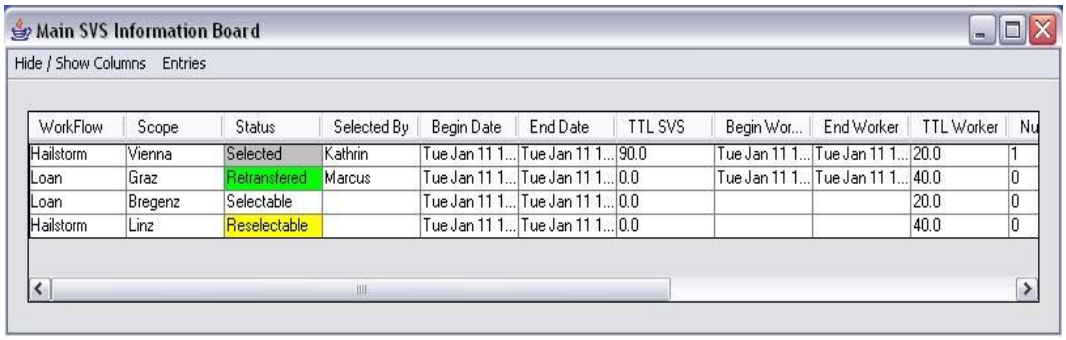
Basically there exist two kinds of processes: the worker and the master process or according to the CORSO P/C pattern [2] a consumer respectively a producer process.

In contrast to the P/C pattern the worker's job, to fetch a package, is not predefined. It is allowed to choose any package at will. The CORSO space can be seen as a pool that stores and publishes all packages marked as unfinished and the worker can grab any available package it wants at any time.

The master process is the one that fills the pool with new packages. This is done by placing the *WFPC* objects into the *RootWFP* notification list. This list is also accessible by the worker process that will be immediately informed about the newly added package. If the worker process wishes to execute a content of a package, the appropriate *WFPC* and *WFC* objects are copied (III.2.4) into the local *RootWFP* notification list of the worker process (on the left side of Figure 8). In order to get the proper package, the worker process has to place a request into its *WCRC* object (Figure 14 and Figure 18). The master process reads the object, performs the task and writes the result into the *WCAC* that can be used for further processing by the worker.

The master process is also the one that may remove any communication object as well. A package can be removed only if it is done. If the package is still selected by a worker the master process has to throw an exception. If the master process wants to remove unauthorized workers it has to delete the *WCC*, *WCRC*, *WCLC*, *WCAC* objects and update the *WCCs*.

Additionally the master process offers controlling and monitoring functions. Monitoring a communication object is supported by SVSDM by calling CORSO's notification service. Any change in the pool is reported immediately and is



WorkFlow	Scope	Status	Selected By	Begin Date	End Date	TTL SVS	Begin Wor...	End Worker	TTL Worker	Nu
Hailstorm	Vienna	Selected	Kathrin	Tue Jan 11 1...	Tue Jan 11 1...	90.0	Tue Jan 11 1...	Tue Jan 11 1...	20.0	1
Loan	Graz	Retransfered	Marcus	Tue Jan 11 1...	Tue Jan 11 1...	0.0	Tue Jan 11 1...	Tue Jan 11 1...	40.0	0
Loan	Bregenz	Selectable		Tue Jan 11 1...	Tue Jan 11 1...	0.0			20.0	0
Hailstorm	Linz	Reselectable		Tue Jan 11 1...	Tue Jan 11 1...	0.0			40.0	0

Figure 19: information board

presented in a table (Figure 19, see also ch. III.4.2) accessible for authorized users only. The *"Main SVS Information Board"* presents the values the *WFPC* and *WFC* objects have stored. It shows the name of the *WFC* object, the name of the *WFPC* object, the status of the package, the name of the worker who fetched it, when the package was put into space and when the status of it becomes expired. Furthermore it illustrates the amount of time in minutes the package may spend in the space (*TTL SVS* ch III.5) and when a worker fetched the package. Additionally it shows the deadline when the worker has to be finished with execution and how long she/he has time to do so. It presents how many

files are contained in the *WFPC*, if there is already an answer, what is the title of it and how many files were sent back.

III.2.4. Communication between Master and Worker Spaces

Whenever the master process is started, it creates and connects to a CORSO space on its local machine. The same thing happens when the worker process is executed. This is a great thing because it has now all the services CORSO

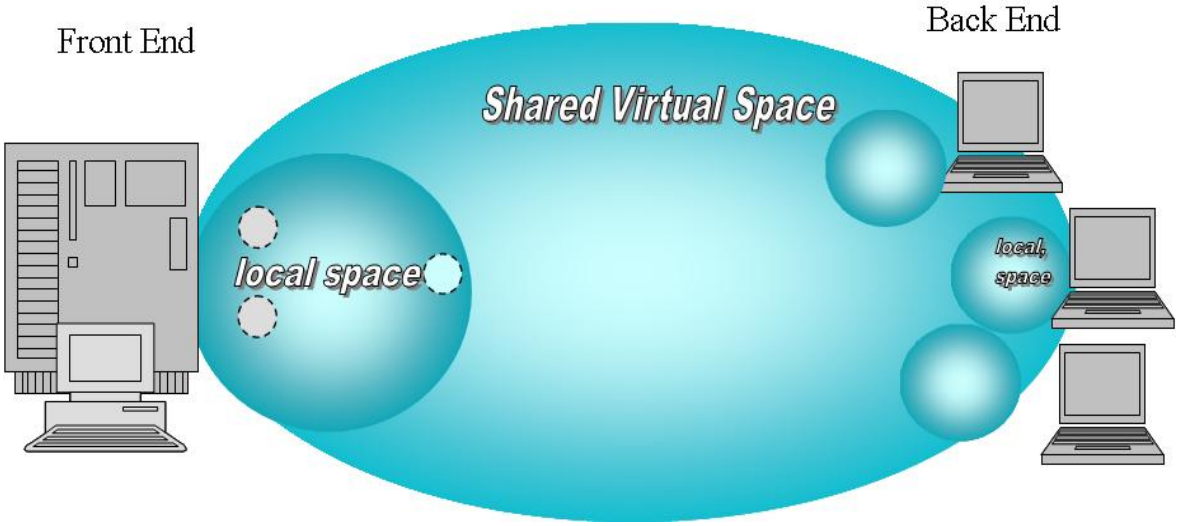


Figure 20: local spaces

can offer. The worker space is now able to perform its tasks stored in the local cache, but it cannot see the packages stored in the local master space. First, the worker has to create a global space with the master space in order to fetch the information about available packages. After connecting to the master space, a global space has been created and all shared objects of the master space are

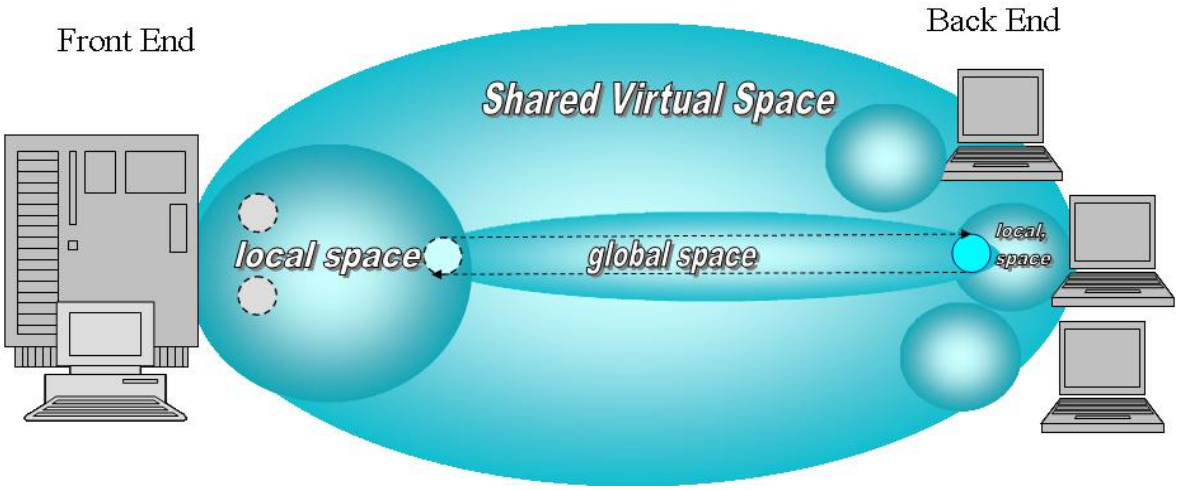


Figure 21: local spaces merged to a global one

replicated according to the used distribution strategy to the local space of the worker. Note that the primary copy still remains at the master process site. This means that the first step in building up the object tree has been made. The agent is now able to choose a package and ask for it by writing the appropriate request into its communication object, into the worker process' *WCRC*.

In case of intended selection of a package the worker process has to perform some preparations first. First of all it has to obtain the *OID* of the requested object and has to read it in a non transactional way. This is necessary otherwise the primary copy would come to the worker's site. Any other reading operations, for instance of the workflow, have to be done in the same way. These two steps make sure that copies of the objects do exist in the local cache. The next task is to create a clone of these objects and to store them in the local space of the worker. These newly created objects also receive a completely new *OID*. This would not be necessary, if there were only read operations made on this object in the future. But values have to be changed and therefore also written. The introduction into CORSO showed [II.2] that in order to be able to write a value into a communication object the primary copy of it has to be owned first. Since the worker is not allowed to obtain the primary copy for reasons already mentioned, it has to be satisfied with the clone of it which has been just created locally. The worker process is now able to perform as many online or even offline writing operations as required without concerning any one else.

The easiest way of communication between the two processes would be the following. By writing the request into the shared object the master process is notified by CORSO immediately and starts a new thread. There it performs the request while the worker process is waiting with a synchronous reading of the answer object. After the master process has performed the requested task it writes the result into the *WCAC* of the worker which is read by the worker process. The problem is that the master process does not know when the workers go offline. Meaning that it has performed a task and for instance marked a package selected but the worker did not receive the result because it went offline in the meantime. So, the package is not really fetched and cannot be selected either until the deadline expires.

The solution would be to let the worker write an acknowledgement. If the master process receives it via synchronous reading of the *WCRC,* then it knows that the result has been read and the selected package is really used. The problem coming up is: how does the worker know if the acknowledgement has been really received? This is the so called two-army problem [1, pp. 372].

The two-army problem shows the difficulty of getting an agreement between two processes. To describe the problem in the best way let us assume that there is a battle starting between two armies. The black army has about 5000 troops while the white 6000, but split into two. The main point is that the white army can only succeed about the black one if both parts can coordinate their attack. If either one attacks the black army alone it will lose. Furthermore the two parts of the white army can only communicate with each other by using an unreliable communication channel – messengers who can be captured by the black army. Let's assume that General 1 (G1) of the first part of the white army sends a message to General 2 (G2) of the second part of the white army telling him to attack in three hours. G2 agrees with the message and sends an acknowledgement back to G1. G1 receives it but G1 thinks that G2 might think that G1 has not received the acknowledgement and so G2 is not daring to attack. So G1 gives an order to send another acknowledgment back to G2 saying that G1 has received G2's message. The point is that G2 has the same doubts G1 had before. However it is easy to see that this game can be played forever because the sender of the last message does not know if the last message has arrived and so he will not attack. Furthermore the receiver of the last message knows that the sender is not sure meaning that he will not risk certain death either.

This shows that just synchronous reading of the shared object is not sufficient enough and that it has to be searched for another solution to solve the problem. The solution is to use acceptors, another reliable service offered by CORSO.

Acceptors are CORSO notification objects. Each time a *getNameConstOid* is retrieved, the name of the acceptor passed as an argument, a new *CONST OID* is created and inserted into the acceptor's notification and returned to the application. Whenever an object is written into that *CONST OID* the notification

fires at the site, where the acceptor was created. This means that whenever the worker receives the acknowledgement or the result from the master process it also gets the name of an acceptor created for that particular communication session. The only task left for the worker to perform is to write another acknowledgment into the *CONST OID*. If the master process receives the positive message within a given period of time the request made by the worker is seen as successfully committed otherwise all transactions are rolled back.

This complicated procedure is important because it is necessary to keep all primary copies of every single package in the master space. If a package is fetched by a worker, it needs to be marked as selected. This has to be done by the master process. If it was done by the worker itself, the primary copy of the package would move to the worker's space. This is not acceptable because the worker could go offline forever meaning that the package is lost forever as well since the primary copy of it can never be fetched and so never be distributed ever again.

III.2.5. Distribution Strategies

An important topic in the world of CORSO is how objects are handled. In the introduction into CORSO it was explained that there are several points the developer has to take care of. The first one is reliability, the second propagation and the third concerns the way of reading an object.

1. Reliability: The SVSDM prototype uses reliability class 1. Although a lot of files could be saved in databases the shared objects cannot. They have to be placed in the virtual space where they have to stay as long as they are removed. This action so far has to be done manually but the point is that the packages have to stay visible after system failures as well. If class 0 was used a single failure of the master process would cause loss of data.

2. Propagation: The SVSDM prototype has the Lazy Propagation implemented. This means that objects are propagated only at explicit read requests. Actually these characteristics do only affect the WorkFlowContainer objects. *WFPC* objects are replicated to every site immediately since the information board there has to present data. The

question is when the according WFC object, a sub object of a package, has to be replicated. This should be only the case when the user explicitly whishes to perform the content of the package meaning when the user places the appropriate request into his WCRC.

3. Reading: Since Read-Main is not yet fully supported by CORSO, SVSDM has to use the Read-Next feature.

III.3. SVSDM API

This chapter explains how to use the objects that are important for correct execution of SVSDM. The three most important objects, that are declared as public for the user, are *WorkFlowPackageContainer* that represents the envelope of the package, the object *ModificationOfNotification* that inserts the *WFPC* into the notification vector of the master process respectively its counterpart at the worker site. It is important to mention that these three objects do not and would not be able to do the entire distribution work on their own. It is a combination of different services and other objects and threads that are invisible for the user – most of them already mentioned in the previous chapters. See Appendix for full API description of the following objects.

III.3.1. Object WorkFlowPackageContainer

The *WorkFlowPackageContainer* is the object that is taken for distribution and for monitoring purposes. It is an object that is replicated and shared between several sites. This means, it has to be small in order to avoid too much data traffic and to enable fast communication.
At this point it should be mentioned that storing additional data (*instancefiles*) for further description of the package is not the best solution. It would be better to store this data in another object referenced by an *OID* the same way it is done with the content of the package. See chapter III.6.2 for more details.

For the distribution of the object it is only important to know how the status of it looks like. Is the package already selected or still free to be fetched? The object's public method *getStatus():String* is accessed by both the master and the worker spaces to get this information. On behalf of this information the package is either shown on the information board or not. In case of the master process the status of the package does not play a role, because it has to show every package regardless of its status. The worker process is only allowed to present the user packages that are free to be selected. Any other is hidden from the user.

Monitoring characteristics of SVSDM are also built into this object. Apart from the ability to be informed about who has selected the package, using *getselectedby(),* it is also important to know when the package has been fetched, but most importantly when it has to be redelivered at the latest. Actually there are two deadlines. By calling *timeToExpireCorso()* the user gets the amount of time that is left before the package has to leave the space. If the package is allowed to stay there forever, then an exception is thrown, so it would be better to call *isForeverInSVS()* first. The package will not be removed automatically after it has expired. The status changes to *Expired* and it can not be selected any more, meaning that the user of the master process has the responsibility to remove it from the space manually.

In the first version of SVSDM there was a thread that did deletion automatically and then just informed the user about this action. This method has been removed to give back the user full control. The only thing this thread does now is to check whether a package has expired or not.

III.3.2. Object ModificationOfNotification

This object inserts, removes or updates both *WFPC* and *WFC* objects. This class offers the possibility to search for packages and authorized users whose communication containers (*WCC*, *WCRC*, *WCAC* and *WCLC*) are also placed into the space by means of this object. It creates, destroys and reestablishes connections to the CORSO space. It can change and define strategies and create or search for named objects. It can be said that this object represents the basic working layer of SVSDM.

III.3.3. Object ModificationOfNotificationLocal

The three most important methods this class offers are *moveFromLocalToServer()*, *moveFromServerToLocal()* and *transferToBPEL4J()*. The first enables the redelivery of a selected package. It should be noticed that the method does not check whether the package has an answered stored or not. The second method tries to fetch a selectable package on behalf of the user. The last one executes the workflow and starts the graphical User Interface of the appended application.

Apart from these three methods *ModificationOfNotificationLocal* offers more or less the same methods as *ModificationOfNotification,* but its methods concern storage and administration of the objects stored in the local space of the worker.

The following threads represent an important part in the architecture of SVSDM:

- Thread WorkerCommunicationThread

  This thread is responsible for the communication between the master process and all other worker processes. Every time the CORSO notification fires, another thread is started that performs the request. A request can be one of the followings:

  - **Log**: this flag tells the master process that new log data is coming. The informing request has to start with a number that indicates how many log entries have to be added. A log entry consists of three parts: *"code_when_what"*. The code tells the process what kind of entry it is, when indicates the time of occurrence while what can store additional describing information. See also description to *WCLC* in ch. III.2.1.
  - **Select**: the worker process would like to fetch a package and mark it as selected. For this reason it has to transmit the *OID* of the requested package and the user's name.
  - **Release**: this flag indicates that the owner of the package would like to get rid of it. This could mean that the user has selected a package by mistake or she/he realizes that the deadline cannot be held. The *OID* of the package that should be unselected is needed.
  - **Setanswer**: The user has executed the content of the package and would like to upload the answer. The master process takes it and writes it into the appropriate object. Meanwhile the object's status changes from *Selected* to R*etransferred*. In order to perform this request the master process needs the *OID* of the package, the title of the answer and the answer itself in bytes.

  Any other types of requests are meant to be an error and refused.

- Thread WorkFlowPackageBoardFrame

  This thread is the information board for the user of the master process. It shows all existing packages that are currently in the space. As seen in Figure 19 all data the *WFPC* object is able to offer is shown. This helps the user to be up to date. The board currently represents the only possibility to remove a package from the space. The user just has to click on the row and press the *Del* button. Everything else is done transactional secure in the background.

- Thread tableThreadServer

  This thread handles the information board of the global space at the worker's site. It shows the user all packages that are selectable. In contrast to the information board at the master process' site the board shows user relevant pieces of information. This would be for instance how long the package is available and for how long it can be obtained by the worker.

- Thread tableThreadLocal

  This is another information board run by the worker process but this one is concerned with the packages stored in its local space. The more important data this table needs to publish, would be how much time is left until the package expires or how many answer have been made so far.

III.4. Implementation

So far it has been reported about the structure and processing of the SVSDM prototype. This chapter shows the functionality in a programming language way in order to demonstrate, on the one hand how SVSDM is implemented and on the other hand how easy it is to work with CORSO.

III.4.1. Authorization and Communication

Authorization of workers is achieved by creating objects necessary to implement the Request / Answer pattern. These objects are put into the space whenever the user of the master process adds a new agent to the system. At the same time these objects are used to enable communication between the master and the worker processes.

The participating objects for authorization would be *WCC* and *WCCs* and the main ones for communication *WCRC* and *WCAC*. According to Figure 8 the object IDs of *WCRC* and *WCAC* are inserted into a notification list and the *OID* of *WCC* added into *WCCs*.

Before anything else can be done the root objects have to be found first. CORSO offers named objects for this purpose. The names of the notification lists are *contnameCOMR* and *contnameCOMA*. The one for *WCCs* is *RootWCCs*. They are found by means of the public method *setContainerOids(boolean createOid)* provided by the *ModificationOfNotification* class that in turn calls

```
public void setContainerOidForCOMR(boolean createOid) throws Exception {
  this.ContainerOidCOMR = Utils.getOidForRoot(this.getCorsoConnection(),
                                              this.contnameCOMR,
                                              this.getConnectionSite(),
                                              this.getConnectionStrategy(),
                                              createOid);
}
```

setContainerOidCOMR(boolean createOid) and respectively the other methods for the other two communication *OIDs*. The Boolean variable indicates if the shared objects should be created if they do not exist. If they are not present and the variable is set to *false,* an exception is thrown.

Once the method has run successfully the existing notification objects have to be read in. To achieve this goal a CorsoNotification object, *RootCOMR*, has to be instantiated first. Then the application has to ask CORSO to read the shared object with the received *OID ContainerOidCOMR* and to put its values into the newly created object.

```
CorsoTopTransaction tx = this.getTopTrans();
CorsoNotification RootCOMR = new CorsoNotification();
this.ContainerOidCOMR.readShareable(this.RootCOMR, tx, CorsoConnection.NO_TIMEOUT);
```

The next step would be to insert the *WCAC*, *WCRC* and *WCC* objects into the right place. This is done with the public method *placeNewWCC(String name)*. *Name* indicates the name of the worker who should be added and is used for verification of authorization. The first part of the method checks whether the

```
public void placeNewWCC(String name) throws Exception {
   // check if user already exists
   if (this.isExistingWCC(name) == true) { throw new Exception("User already exists"); }
```

user exists or not. The second part is concerned with the creation of a Corso*TopTransaction* and the new *OID*s for each object.

```
CorsoTopTransaction top = this.getTopTrans();
CorsoVarOid WCCOid = this.getCorsoConnection().createVarOid(this.getConnectionStrategy());
CorsoVarOid LogOid = this.getCorsoConnection().createVarOid(this.getConnectionStrategy());
CorsoVarOid AnsOid = this.getCorsoConnection().createVarOid(this.getConnectionStrategy());
CorsoVarOid ReqOid = this.getCorsoConnection().createVarOid(this.getConnectionStrategy());
```

In the third step the required objects are created, initialized and written into the space. It can be seen that everything happens within a transaction. This is necessary since unused or unreferenced objects in the space should be avoided. Let us assume CORSO is able to write the first two objects, but has problems to do so with the left ones. In case of a transaction, either all objects are written into the space or none. This makes work easier and avoids that unused object IDs cruise in the space.

```
WorkerCommunicationLogContainer newlog =
        new WorkerCommunicationLogContainer(LogOid, name);
LogOid.writeShareable(newlog, top);
WorkerCommunicationContainer newcont =
        new WorkerCommunicationContainer(WCCOid, AnsOid, ReqOid, name, LogOid);
WCCOid.writeShareable(newcont, top);
WorkerCommunicationAnswerContainer newans =
        new WorkerCommunicationAnswerContainer(AnsOid, WCCOid, ReqOid);
AnsOid.writeShareable(newans, top);
WorkerCommunicationRequestContainer newreq =
        new WorkerCommunicationRequestContainer(ReqOid, WCCOid, AnsOid);
ReqOid.writeShareable(newreq, top);
```

In the fourth part of the method, the *OIDs* of the created shared objects need to be inserted into the notification list. *CorsoNotificationItems* have to be created and placed in the *CorsoNotification* object. Finally, the last object, *WCC*, has to be added to the *WCCs*. This means to open the shared object, add the *OID* and

```
CorsoNotificationItem newitema = new CorsoNotificationItem(AnsOid,
   CorsoNotificationItem.INITIALIZE_WITH_CURRENT_TIMESTAMP,
   CorsoNotificationItem.CURRENT_TIMESTAMP);
CorsoNotificationItem newitemr = new CorsoNotificationItem(ReqOid,
   CorsoNotificationItem.INITIALIZE_WITH_CURRENT_TIMESTAMP,
   CorsoNotificationItem.CURRENT_TIMESTAMP);

this.RootCOMA.addItem(newitema, top);
this.RootCOMR.addItem(newitemr, top);
```

to write the new object back to space. The very last step is to commit the

```
WorkerCommunicationContainers conts = new WorkerCommunicationContainers(this.getContainerOidCOM(), top);
conts.addWCCOid(WCCOid);
this.getContainerOidCOM().writeShareable(conts, top);
top.commit(Utils.TimeOutForTransactions);
}
```

transaction which also ends the Java method. If commitment fails, the exception is just forwarded back to the calling method.

As mentioned before these objects are meant for communication between the master and worker process and for authorizing users for that. How objects for the communication come into existence has been just explained. The remaining question is how authorization is done. This policy is performed on the client side

```
private void getMyReqContainer() throws CorsoException, Exception {
  // get OID of the WCCs
  CorsoVarOid contsOid = this.conex.getNamedVarOid(Utils.NameOfContainerForRootWCCs,
                                                   modnot.getServerSite(), null, false,
                                                    Utils.TimeOutOfWorkerWaitingForServer);
  // read the object from the space
  conts = new WorkerCommunicationContainers(contsOid,null);

  // get every OID contained there and perform search by going through the vector
  Vector all = conts.getWCCs();
  for (int i=0; i<all.size();i++) {
    // take OID from the vector and open WCC object
    CorsoVarOid tmp = (CorsoVarOid) all.elementAt(i);
    WorkerCommunicationContainer cont = new WorkerCommunicationContainer(tmp,null);
    // if name equals in both objects the user is authorized for communication otherwise do not do anything
    if (cont.getName().equals(modnot.getConnectionUser())) {
      this.reqcontOid = cont.getReqOid();
      break;
    }
  }
}
```

by calling the method *getMyReqContainer*. Whenever the local space is merged with the global one, the client process starts a search for its assigned *WCRC*. This is done via reading the named object *WCCs* and going through its entries. If the *OID* is found, then it is stored in the global variable *reqcontOid*. Otherwise this variable stays uninitialized, which subsequently means that no requests can be placed.

III.4.2. Information Board

The information board is the interface between the user and the system. It shows the user what is going on in the shared virtual space. In the prototype there are four information boards implemented. Three of them handle distributed packages while the last one is concerned with the presentation of workflows only. Basically all four of them are implemented the same way. The only difference they have is the name of the shared objects that keep the *CorsoNotification* and the restriction rules. Rules in terms of what kind of entries are allowed to be shown and which ones are prohibited. In the following the information board of the master process that shows all packages is taken as a representative for the other three and explained in more detail.

As in the previous chapter each of the notification lists is stored in a named object. This has to be found first in order to receive the object's *OID*. This *OID* is then used to read the value of the object, a *CorsoNotification*. Actually it is not

```
ContainerOidWFP = Utils.getOidForRoot(conex, Utils.NameOfContainerForRootWFP,
                        modnot.getConnectionSite(), modnot.getConnectionStrategy(),
                        false);

CorsoTopTransaction txnotif = conex.createTopTransaction();
this.RootWFP = new CorsoNotification();
ContainerOidWFP.readShareable(this.RootWFP, txnotif, CorsoConnection.NO_TIMEOUT);
txnotif.commit(Utils.TimeOutForTransactions);
myvector = new Vector();
```

necessary to execute these commands within a transaction, but it would make sure the primary copy is still obtained by the master process in case of buggy coding.

The next step is to start the notification service. The service has a timeout set in order to avoid blocking of the thread it is running in. Once the service has fired, meaning that an observed object has changed its values, a transaction is started and the object is read. Transaction is necessary because the *valid* variable of the object might have been set to *false*. In that case deletion has to be performed within the same transaction.

```
CorsoNotificationItem fired = null;
fired = this.RootWFP.start(Utils.TimeOutForNotification, null);
if (fired != null) {
  CorsoVarOid firedoid = fired.varOid();
  CorsoTopTransaction tx = conex.createTopTransaction();
  WorkFlowPackageContainer struct = new WorkFlowPackageContainer(firedoid,tx);
```

If the object that has been read is already in *myvector,* then the object values have changed, otherwise a new object was added to the notification list. The first if-statement checks this case. Within that it has to be checked, if the new object is still valid. This is necessary, because the system might have crashed and the table is currently being built up. If the object is valid, it is inserted into the

```
if (!myvector.contains(fired)) {
        if (struct.isValid()) {
          // if object is valid add it to vector and present it in the information board
          myvector.addElement(fired);
          Utils.insertIntoMyTable(table,struct);
        }
        else {
          // object is invalid meaning that it needs to de deleted
          if (!this.deleteWorkFlowPackage(firedoid,fired,tx)) {
            // deletion failed for whatever reasons, put it into the vector
            myvector.addElement(fired);
            tx.abort();
            continue;
          }
        }
        tx.commit(Utils.TimeOutForTransactions);
      }
```

table. Otherwise *deleteWorkFlowPackage()* tries to remove the object from the space. If it did go well, the transaction is committed and the service waits for the next object to fire. If something went wrong the object is added to *myvector,* but it is not added to the table since it is not valid.

If the changed object is in the vector, the specific table entry has to be updated. This could mean either to display the changed values or that the row has to be removed from there.

```
if (struct.isValid()) {
    // update table because valid object has changed its values
    Utils.updateMyTable(table,struct);
}
else {
  // object has expired => free and delete
  if (this.deleteWorkFlowPackage(firedoid,fired,tx)) {
     // deleted successfully => remove object from the information board
    if (!Utils.removeRowFromMyTable(table, struct)) {
      // could not remove row from table => update table and abort transaction
      Utils.writeToSystemLine(deb,"Failed to remove line from Table");
      Utils.updateMyTable(table,struct);
      tx.abort();
      continue;
    }
  }
  else {
    // deletion failed => update table in order to show user invalid object
    Utils.updateMyTable(table,struct);
    tx.abort();
    continue;
  }
}
tx.commit(Utils.TimeOutForTransactions);
```

The if-statement starts again with the *isValid()* method call. If the object is still valid then the changes made to the object are shown on the table and the transaction is committed. The more difficult part is to delete the package. If deletion fails the table is just updated and the transaction is not committed. This is necessary since in order to remove a package several steps have to be made. This means that the *OID* of the package has to be deleted and that the NotificationItem has to be removed.

If the object could be deleted successfully, the thread tries to remove the entry from the table. If this fails the table is updated only. All in all the transaction is only committed, if first the object can be removed from the space and second if the appropriate row from the table disappeared.

III.4.3. WorkerCommunicationThread

In the following the implementation of the communication protocol between the master and the worker process is described. As the file name already indicates, this thread waits for incoming requests. Requests can be placed in the *WCRC* object of the appropriate worker. This means that the communication thread has to be notified whenever a change happens on one of these objects. Similarly to a web server, the only task of that thread is to listen to changes, specify the

```
CorsoNotificationItem fired = null;
fired = this.RootCOMR.start(Utils.TimeOutForNotification, null);
if (fired != null) {
  if (vect.contains(fired.varOid())) {
    // WCRC object has changed => start new thread to perform request
    RequestHandlerThread reqhandler =
        new RequestHandlerThread(fired.varOid(), acceptors, modnot);
    reqhandler.start();
  }
  else {
    // add newly created WCRC into the vector
    vect.addElement(fired.varOid());
  }
}
```

object that has been changed, to start a new thread *reqhandler* that will perform the request and to keep on listening. The variable *vect* is necessary to know whether the *WCRC* object has changed or has been newly added. If it has been inserted into the notification list recently, then it cannot be in *vect* and so there is no need to start a new thread.

The newly started thread's task is to response to the request. For this reason it has to know what kind of request it is. This can be found out by means of the method call *getMSGID()*. This procedure returns a number that stands for a specific request. The two most important requests are *select* and *setanswer*.

If a worker process would like to select a package, its request has to maintain several different types of information. First of all the master process needs to know the *OID* of the package and second it needs the name of the agent who

would like to work with it. If all these data are given, the thread opens the package and checks if it has expired or has already been selected. If the package

```
case WorkerCommunicationRequestContainer.MSGID_SELECT:
        try {
          // get OID of the requested package and the name of the user
          CorsoVarOid tmp = (CorsoVarOid) data.elementAt(0);
          String name = (String) data.elementAt(1);

          // check if WFPC is not selected and still free to be fetched
          WorkFlowPackageContainer wfp = new WorkFlowPackageContainer(tmp,top);
          if (!wfp.isSelected() && !wfp.isExpiredCorso()) {
            // send back acceptor to confirm mission and to make sure there is no loss of connection
            this.sendAnswerAndCheck();

            // so, everything went well - mark WFPC selected
            wfp.setSelectedBy(name);
            tmp.writeShareable(wfp, top);
          }
          else {
             // package cannot be selected => write Error message into WCAC object
            if (wfp.isSelected()) {
              this.placeAnswer(reqcont.getAnsOid(),"
                                Error – WorkFlow already selected by another one",reqcont.getReqNr(),top);
            }
            else {
              this.placeAnswer(reqcont.getAnsOid(),"
                                Error - WorkFlow expired in the meantime",reqcont.getReqNr(),top);
            }
          }
          break;
        }
        catch (Exception eselect) {
          // internal error occurred, for instance could not create a top-transaction => inform worker
          this.placeAnswer(reqcont.getAnsOid(),"Internal Server Error - code 578",reqcont.getReqNr(),null);
          this.removeAcceptorName(acceptorname);
          return;
        }
```

is free, the master process calls first a method called *sendAnswerAndCheck()*. This is the procedure that tries to confront the two army problem by placing the name of an acceptor in the worker process' *WCAC* and by waiting for a valid

response. If the response is received on time the package is marked as selected the new object is written back into the space. This will cause that each information board has to update its tables, more exactly it will cause that the just selected package disappears from their tables. It is important to mention that every single step so far has been done within a single top transaction. Even the *sendAnswerAndCheck()* method is run in a sub transaction of it. This means that if one link in the chain breaks, not a single step is committed. In any other case an error message is placed in the *WCAC* object and the request is performed.

Setanswer on the other hand is a request that would like to publish the result

```
case WorkerCommunicationRequestContainer.MSGID_SETANSWER:
    try {
      // get OID of the package where answer should be placed, get title of answer and its files
      CorsoVarOid tmp = (CorsoVarOid) data.elementAt(0);
      String title = (String) data.elementAt(1);
      byte[] files = (byte[])data.elementAt(2);
      // open WFPC object where answer should be stored
      WorkFlowPackageContainer wfp = new WorkFlowPackageContainer(tmp,top);
      WorkerCommunicationContainer cont = new WorkerCommunicationContainer(reqcont.getParent(),top);
```

for the content of a package. In order to perform the request, the master process demands the *OID* of the package, the title of the answer and the zipped file. The next step is to open the *WFPC* and the *WCC* of the worker process. This is necessary, because two things have to be checked first. A package can only receive an answer, if it is either *Reselectable* or selected by the requesting

```
if (wfp.getStatus().equals(WorkFlowPackageContainer.StatusReSelectable) ||
    cont.getName().equals(wfp.getSelectedBy())) {
  // send back acceptor to confirm mission and to make sure there is no loss of connection
  this.sendAnswerAndCheck();
 // write status, name of the worker and of course the answer into the WFPC object
  wfp.setStatus(WorkFlowPackageContainer.StatusReTransfered);
  if (!wfp.isSelected()) { wfp.setSelectedBy(cont.getName());}
  wfp.setAnswer(title,files);
  tmp.writeShareable(wfp,top);
 }
```

worker. If it is the case, for a confirmation is asked for and the request is

performed. Additionally, in case the package's status is *Reselectable,* the name of the agent has to be set as well.

Any other kind of combinations is seen as an error and reported back to the worker. For instance the answer might have come too late. The worker forgot

```
else if (wfp.isExpiredCorso()) {
  // no response is accepted since object is expired
  WorkerCommunicationAnswerContainer anscont = new
      WorkerCommunicationAnswerContainer(reqcont.getAnsOid(), top);
  anscont.setAnswer("REFUSED_WorkFlow expired in the meantime",reqcont.getReqNr());
  reqcont.getAnsOid().writeShareable(anscont, top);
}
else if (wfp.isSelected() && !cont.getName().equals(wfp.getSelectedBy())) {
  // no response is accepted because the one who has a copy of the package and the one who wants
  // to response do not match
  WorkerCommunicationAnswerContainer anscont = new
      WorkerCommunicationAnswerContainer(reqcont.getAnsOid(), top);
  anscont.setAnswer("REFUSED_WorkFlow already selected by another one",reqcont.getReqNr());
  reqcont.getAnsOid().writeShareable(anscont, top);
}
else {
  // response is not accepted because there is already one
  WorkerCommunicationAnswerContainer anscont = new
      WorkerCommunicationAnswerContainer(reqcont.getAnsOid(), top);
  anscont.setAnswer("REFUSED_WorkFlow has been completed by another worker",reqcont.getReqNr());
  reqcont.getAnsOid().writeShareable(anscont, top);
}
```

that a package can expire but still wants to redeliver the answer. Another possibility is that the package is selected by someone else. This might happen if the package was held by the worker for too long without redelivering an answer. This indicates the master process to republish the package that might have been selected again by another worker. The privilege to answer the content of the package is obtained either by the worker who has the package or in case the package status is *Reselectable* by the worker who had the package last. The last possibility is that the package already contains an answer provided by another worker.

III.4.4. Fetching a package

Another interesting point from the implementation's point of view is how the worker process fetches a package. If an agent decides to take over work, a lot of

```
CorsoTopTransaction tx = this.getTopTrans();
// get WFC object referenced by the WFPC object wfpcont
WorkFlowContainer wfcont = new WorkFlowContainer(wfpcont.getWorkFlowRef(), null);
// create new oids that are used for the clones
CorsoVarOid wfpoid = conex.createVarOid(this.getConnectionStrategy());
CorsoVarOid wfoid = conex.createVarOid(this.getConnectionStrategy());
// make clones
WorkFlowPackageContainer copywfp = (WorkFlowPackageContainer) wfpcont.clone(wfpoid);
WorkFlowContainer copywf = (WorkFlowContainer) wfcont.clone(wfoid);
// update reference of local copywfp so that it "points" to the copy wfoid
copywfp.setWorkFlowRef(wfoid, copywf.getWorkFlowName());
// prepare request !!
long reqnr = rand.nextLong(); Vector data = new Vector();
// write request !!
data.addElement(wfpcont.getID());  data.addElement(this.getConnectionUser());
reqcont.setRequest(WorkerCommunicationRequestContainer.MSGID_SELECT, data, reqnr);
// send request and confirm it
this.sendRequestAndConfirm(reqcont, tx);
// update setselected of local copy
copywfp.setSelectedBy(this.getConnectionUser());
// write everything into local svs
wfpoid.writeShareable(copywfp, tx);
wfoid.writeShareable(copywf, tx);
// create notitems for the notificationservice
CorsoNotificationItem itemwfp = new CorsoNotificationItem(wfpoid,
    CorsoNotificationItem.INITIALIZE_WITH_CURRENT_TIMESTAMP,
    CorsoNotificationItem.CURRENT_TIMESTAMP);
CorsoNotificationItem itemwf = new CorsoNotificationItem(wfoid,
    CorsoNotificationItem.INITIALIZE_WITH_CURRENT_TIMESTAMP,
    CorsoNotificationItem.CURRENT_TIMESTAMP);
// place them into local RootWFP and commit transaction
this.RootWFP.addItem(itemwfp, tx);
tx.commit(Utils.TimeOutForTransactions);
```

preparation has to be done. First of all it is important that every single step is made within a top transaction except of one issue: the reading of the requested package and all of its sub-objects have to be read without a transaction

otherwise the primary copy of them would be transferred to the host's site and this is not acceptable. Once the objects have been read the process has to clone them meaning that two new *OIDs* have to be created and passed to the *clone()* methods. The next step would be to place the request into the *WCRC*. The method call *sendRequestAndConfirm()* places the request into the object, waits for the name of the acceptor and replies. Things that are remaining is to write the clones into the local space, to put a notification on the package and to place that into the notification list in order to be presented on the user's table. Finally the top transaction has to be committed.

III.4.5. Redelivering a package

In contrast to fetching a package redelivering is a little bit easier. The only decision that has to be done is to check whether there is an answer or not. If there is no answer, then the package has to be released, meaning that other agents get another chance to get the package. Once again a top transaction is

```
CorsoTopTransaction top = this.getTopTrans();
long reqnr = rand.nextLong(); Vector data = new Vector();

// if WFPC loccont has no answer stored then just do setReleaseSelected
if (!loccont.isAnswered()) {
  // according to protocol we have to put into vector WFP's Oid
  data.addElement(loccont.getID());
  // prepare everything for answer
  reqcont.setRequest(WorkerCommunicationRequestContainer.MSGID_RELEASE, data, reqnr);
  this.sendRequestAndConfirm(reqcont, top);
}
```

started and a request with the message ID *MSGID_RELEASE* is placed into *WCRC*. This will induce the server to change the status flag of the package and to clear the *selectedby* variable. Actually the method does not need to evaluate the response of the master process, since it wants to get rid of the package anyway. The only thing remaining is to clear up, removing both the package and all of it sub-objects from the local space.

On the other hand there might be an answer that is wished to be put into the global space. This means that a request object has to be created and put into the *WCRC*. The method *sendRequestAndConfirm()* makes sure that everything

```
else {// otherwise upload answerfiles
  // prepare request
  data.addElement(loccont.getID()); data.addElement(loccont.getAnswerTitle());
  data.addElement(loccont.getAnswerFiles());
  reqcont.setRequest(WorkerCommunicationRequestContainer.MSGID_SETANSWER, data, reqnr);
  try {
    // send request
   this.sendRequestAndConfirm(reqcont, top);
  }
  catch (Exception eback) {
   // check coming response on request – user error or broken network link
   String msg = eback.getMessage();
   if (msg.startsWith("REFUSED")) {
     // answer is refused by master process
     String error = msg.substring(8);
     refused = true;
     refmsg = "Work could be uploaded but was refused at Server site\n" + error;
   }
   else {
     // something wrong with the network
     throw new Exception(eback);
   }
  }   }
// delete WFPC object from local space and commit transaction
loccont.setValid(false);
loccont.getLocalID().writeShareable(loccont, top);
top.commit(Utils.TimeOutForTransactions);
if (refused) {
  // place user error message on screen
  throw new Exception(refmsg);    }
```

goes well otherwise an exception is thrown. In this case it is indeed important to know whether the answer has been accepted or refused by the master process. If the incoming acknowledgment starts with a *REFUSED* then the agent itself made a mistake, for instance the package has already expired and in that case the produced answer becomes irrelevant. An *ERROR* happens if the communication protocol breaks. In that case an exception is thrown otherwise the objects are removed from the space.

III.5. The Use Case Example

The example has been chosen to give a first expression where SVSDM is useful and that developing with CORSO is not really a big deal. Furthermore the development phase of the SVSDM prototype has shown one thing. Implementing a demonstrable GUI of SVSDM takes longer and makes more troubles than writing code for the cooperation of the various communication objects.

The problem the example demonstrates is situated in the area of insurance companies. Let us assume that there was a hailstorm in the southern region around Vienna. After the storm has calmed down the insurance company gets a phone call from one of the assured, *Roland Smith*, who asks the company to check the damages happened e.g. to his house in order to get some money for repair. The worker in the call center, let's call him *Marcus*, receives the personal data of the man and places a request into the shared virtual space by means of the SVSDM. The request that can now be seen by any worker consists of a workflow that describes how to behave and proceed in that case, and of a Java application that executes the workflow. If the profile modus was activated, *Marcus* could also specify that only workers positioned around Vienna are allowed to see the order. It would not make any sense to bother agents working in Graz or Bregenz. The deadline of the package at the agent's site is set to 20 minutes while the package may exist in the space for only 90 minutes. The agent's task now is to grab the order and to gauge the damage. After the work has been done, the agent should upload the report back to the center where the order came from. The result is then extracted out of the package and if the assured's claim is justifiable, he might get some money transferred.

The SVSDM prototype consists of two parts. The first one is the application that represents the master process. The second part, the worker process, is built up of a single window that shows the packages of both the global and local space.

According to the hailstorm problem the description of the SVSDM prototype starts with the master process.

- Master Logon

The Login window has two tasks to fulfill. The first one is that a connection to

Figure 22: logon window of the master process

the CORSO kernel has to be established in order to use SVSDM at all. The user has to specify where the kernel runs and at which port the service can be reached. Filling in user name and password is useful, if access rights are used specifying what a user is allowed to do and what is prohibited. In fact the second task is optional and not a part of the prototype.

If it is clicked on *Connect,* several processes are started in the background first. They have to find out, if the notification lists and the *WCCs* exist. If it is not the case, then these objects have to be created. If one of the process reports an exception an error message pops up and login is refused.

- Control and Configuration Center

After *Marcus* logged on successfully, a window appears showing him the function buttons offered by the prototype. The *Status information* indicates whether there is a connection to the kernel or not.

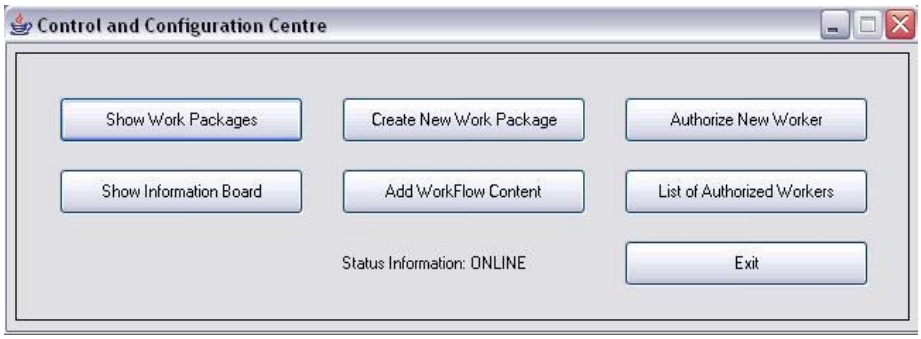


Figure 23: main working window

- New Authorization

Assuming that there is no agent registered yet, *Marcus* clicks on *Authorize New Worker*. This is necessary otherwise communication between the master and the worker process would not be possible.

Figure 24: authorizing new worker

By specifying the name of the agent the communication objects *WCRC*, *WCAC*, *WCLC* are created and inserted into the right notification lists. Furthermore the *WCC* of the worker *Kathrin* is registered in the *WCCs*. All these steps enable agent *Kathrin* to select and fetch packages.

- Add Workflow

The call the assured made was about hailstorms. This means that *Marcus* from the call center has to upload the workflow *hailstorm* and all files attached to it, if they are not already in the space. Actually, storing these workflows in the virtual memory is not necessary. They can be held in a database as well but since this was the easiest and fastest solution so far, CORSO has to serve as a DB.

By pressing the *Save* button a new object *WFC* is created, filled up and stored in the right notification list. The notification list is only used by the master application in order to simulate a database and to be able to update the workflow information board.

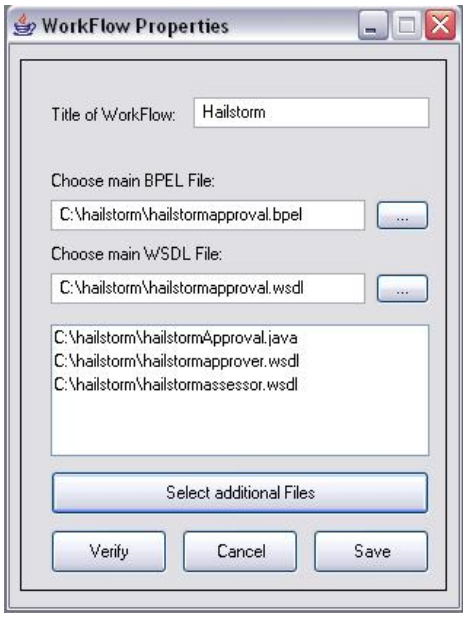


Figure 25: creating a new worflow

- Workflow Information Board

Once the new workflow has been written into the space a new entry appears on the information board immediately. This informs only the user of the master process and no one else. By double-clicking on the row in the table changes on the workflow can be performed.

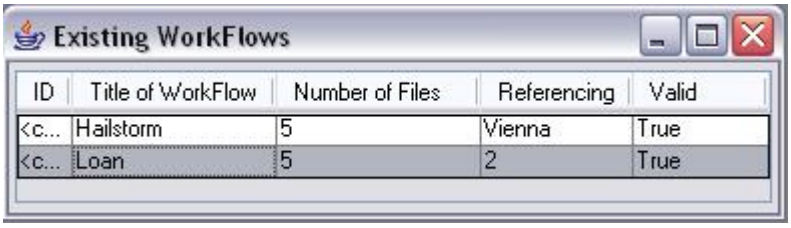


Figure 26: worflow information board

- Add Workflow-package

The steps made so far could have been skipped, if it had been done before. This could mean that the workflow hailstorm might have been used already, for instance in Linz. The following points have to be done and are independent from the previous steps.

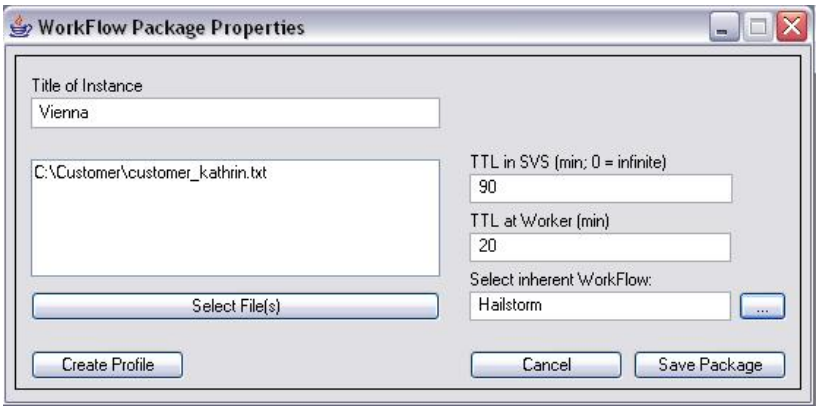


Figure 27: creating a new package

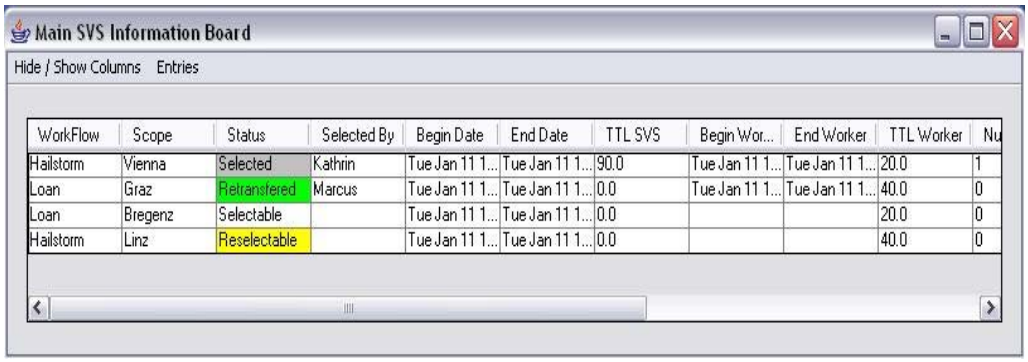
Figure 27 is the window that specifies the characteristics of a package used to distribute the content of it. As it can be seen the title of the package is Vienna since the hailstorm was in Vienna. The additional file *customer_kathrin.txt* might contain information about the assured. The reference to the content of the package, the workflow, is also determined here. Before the package can be saved, *Marcus* has to specify the *time to lease (TTL)* first, the deadlines the agents have to fulfill. *TTL in SVS* stands for the amount of time a package may spend in the space. On the other hand *TTL at Worker* defines the minutes an agent is allowed to work with the package.

- Package Information Board

When the employee of the insurance company clicks on *Save Package,* a lot of things are started to be performed in the background. First of all a *WFPC* object is created and is stored in the notification list. Now the CORSO notification service comes up and notifies all processes everywhere in the network that want to be informed about new objects.

This means that the notification service running in a Java thread fires and reads the object *CorsoData* that contains the object *WFPC*. This object is the one added before. The thread "opens" it and reads the information it needs in order to present the data on the information board.

On the information board of the master process the newly added package has the status *Selectable* and shows every value of each variable of the *WFPC* object. At the "same time" the package appears on the panel (on the right side of Figure 30) of the worker process. The information board there presents only selected values and packages that are either selectable or reselectable.



WorkFlow	Scope	Status	Selected By	Begin Date	End Date	TTL SVS	Begin Wor...	End Worker	TTL Worker	Nu
Hailstorm	Vienna	Selected	Kathrin	Tue Jan 11 1...	Tue Jan 11 1...	90.0	Tue Jan 11 1...	Tue Jan 11 1...	20.0	1
Loan	Graz	Retransfered	Marcus	Tue Jan 11 1...	Tue Jan 11 1...	0.0	Tue Jan 11 1...	Tue Jan 11 1...	40.0	0
Loan	Bregenz	Selectable		Tue Jan 11 1...	Tue Jan 11 1...	0.0			20.0	0
Hailstorm	Linz	Reselectable		Tue Jan 11 1...	Tue Jan 11 1...	0.0			40.0	0

Figure 28: information board showing all packages stored in the space

Coming to the second part of processing, the event of damage where the agent *Kathrin* plays the important role. It is assumed that *Kathrin* owns a notebook or another mobile device that is able to establish a network connection to the internet.

- Worker Logon

Whenever the agent starts the application it has to fill out first the Login

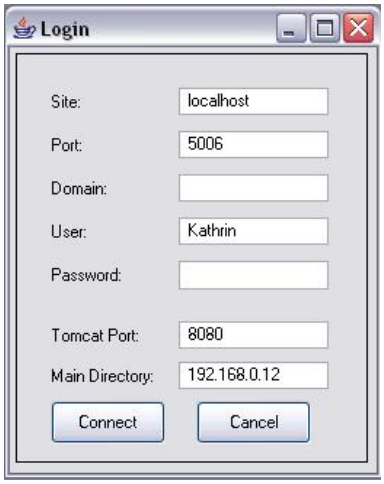


Figure 29: login window for the worker process

window. The agent has to know the site where the CORSO kernel is executed which is in most of the cases the local host and the port on which the kernel is listening. The stated username filled in is used later to find the right objects for communication with the master process.

Since the prototype works with workflows that have to be executed somewhere a *BPEL* engine is running on the agent's machine. This engine is deployed in a

Tomcat Service that listens for requests on a specific port. This number has to be filled in as well in order to be able to perform for instance damage causes.
The *Main Directory* specifies where to look for selectable packages in the space. If the agent is online, a network connection is established, the local space is merged with the global one and CORSO replicates the available packages into the agent's cache.

- Mobile Client Worker Space

The packages free to be selected are presented on the right site of the window. This is the only window the worker sees. If the agent prefers to work offline, the mentioned table stays empty. A newly established connection is recognized automatically and the table is filled up. If a package is fetched by another agent, the entry is removed from the table without any human intervention. That way it is not possible to select a package twice.
The column *workflow* suggests that the package is about hailstorm damages, where *Scope* specifies the area. Since agent *Kathrin* is an expert on that field and currently situated there she grabs the package.
The section on the left shows all packages that are stored in the local space of the agent. Once the user selected and grabbed a package, it will disappear on the right site and inserted into table on the left. This means that agent *Kathrin* has obligated to visit the house of the assured. This action has also changed the status of the package from *Selectable* to *Selected*. Since notifications do report every change this one will not missi either. Exactly at that time the user of the master process is notified (Figure 28) about the fact that agent *Kathrin* has selected the package. This can be seen, because the status has changed, the name of the agent has appeared and the deadlines have been calculated and shown. Any other worker processes are informed about this update as well. Those will notice that the package is selected and therefore they remove the appropriate row from the table

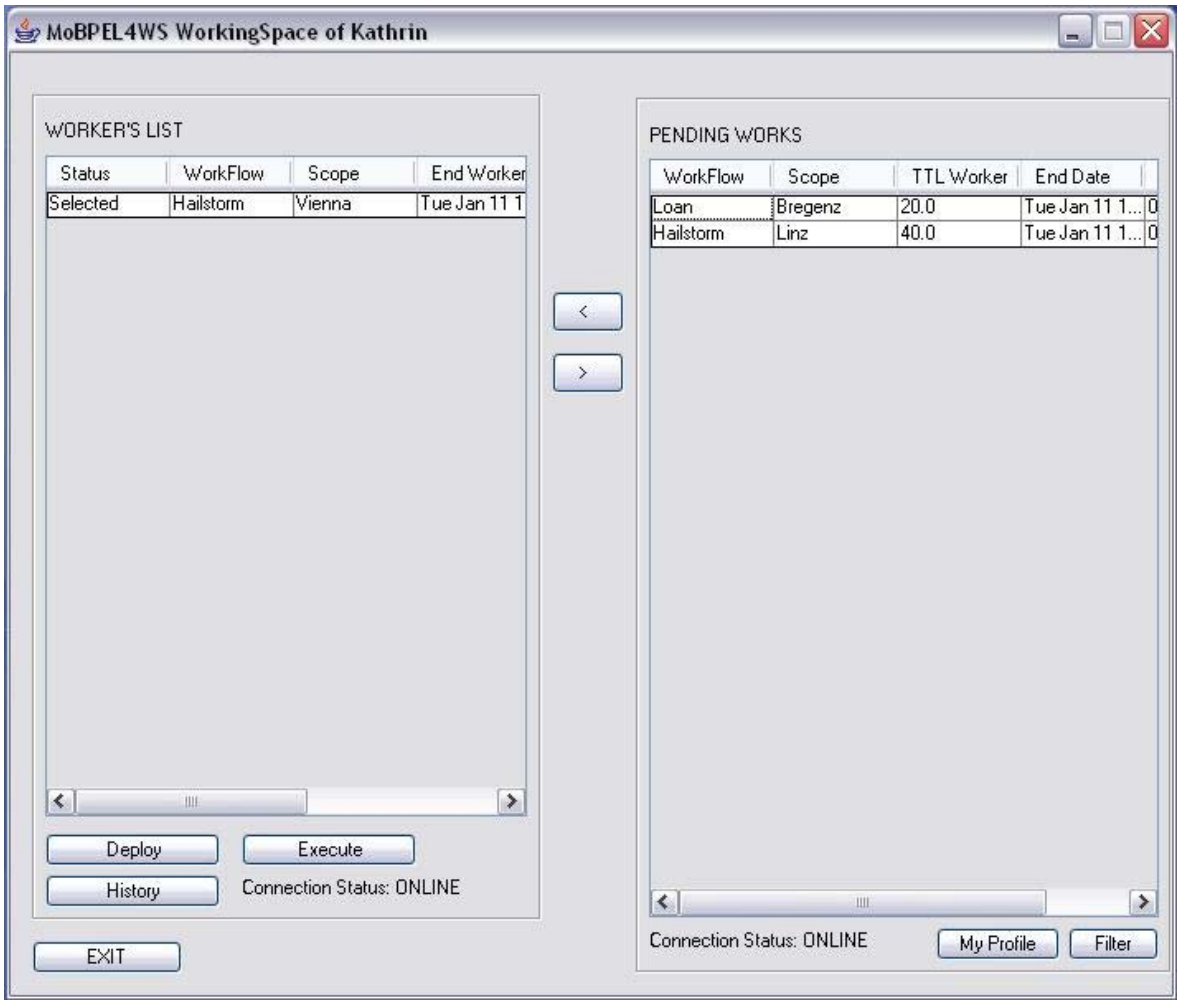


Figure 30: main worker window

Once the package is on the left side, the agent might work with it whenever she wants and both in offline or online modus. The only concern that might come up for the agent is, if she overlooks the deadline. In that case the package appears on the right table once again. This means that the status of the package has changed from *Selected* to *Reselectable* and that it is free to be fetched by any other worker once again. If *Kathrin* tries to get this package once more, she will receive an error message.

Now the agent has to deploy the content of the selected package in Tomcat first. After that *Kathrin* may click on *Execute* to start the Java application. This is a simple GUI that helps *Kathrin* to assess any damages. The used workflow is a very simple one and written only for demonstration purposes. More complicated workflows will require more complex GUIs of course. The utilized workflow takes the amount *Kathrin* thinks the damage will cost, and returns either *Yes* or *No*.

Figure 31: application processing the workflow

When *Kathrin* closes the application its output is stored in the package itself. At the same time the column *Answered* changes to true and the file(s) is (are) stored in the local space.

Kathrin has finished her job, so the package can be uploaded into the global space. This requires the usage of the communication objects *WCRC* and *WCAC* once again. To reduce bandwidth as much as possible only the answer is sent. This array of bytes is then inserted into the primary copy of the package. If the transaction went well the copy of the package is deleted at the local space. At the same time the status of the package changes from *Selected* to *Retransferred* that is also displayed on the main information board.

Marcus' job now is to extract the received information. Actually this could be done automatically as well. At the end SVSDM offers a file that can be used by other services, systems or applications for further processing.

It can be seen that at the starting point the SVSDM prototype was fed with a file. The output is a file as well. This means that SVSDM is a pattern based on CORSO that can be integrated easily into already existing systems.

- Log Table of All Workers

Beside the distribution of workflows, SVSDM also logs actions the agents do while they work with it. Most of the noted data is concerned with packages. It is logged what kind of and how many packages have been fetched and what is the status of the package when the answer is advised to the master process.

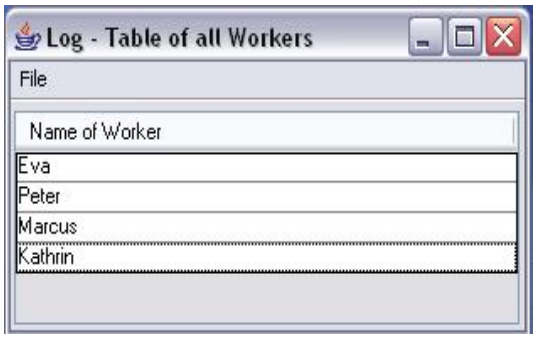


Figure 32: authorized workers

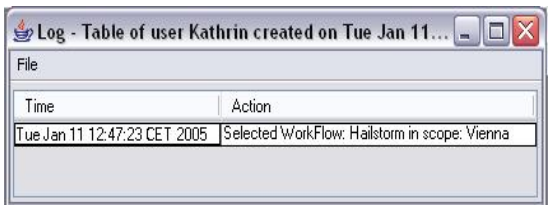


Figure 33: log table of worker *Kathrin*

The things that are not taken into consideration in the course of logging are for instance how often a workflow had to be deployed or how often the Java application has been started. Actually this is not a problem of implementation just one of project description instead.

The two Figures above display the information that could be collected so far for each worker. The window on the left shows all workers currently authorized for communication with the master process. The one on the right side lists all events that have been logged for a single agent.

- Log Table of a Worker

On the agent's site the worker is also able to check its done pensum. The pop

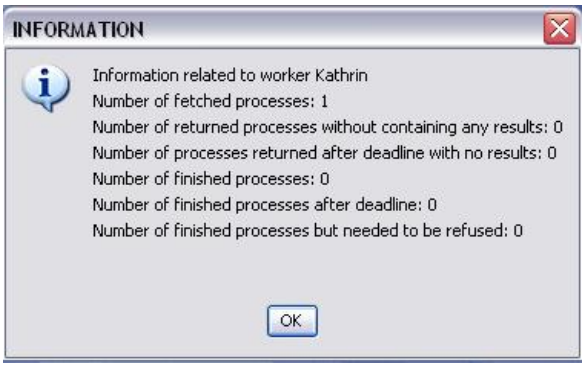


Figure 34: summary of all actions done

up window reports the summary of the data of interest. This offers him the information if the agent is behind schedule or ahead. Furthermore statistics of the collected data could show how efficient the agent works.

- Equipment

The SVSDM prototype was installed and run on different machines. During

Figure 35: Atigo T

the design and implementation phase the prototype was executed on desktop computers. The compiled and tested finale version of SVSDM did run

Figure 36: Insurance worker

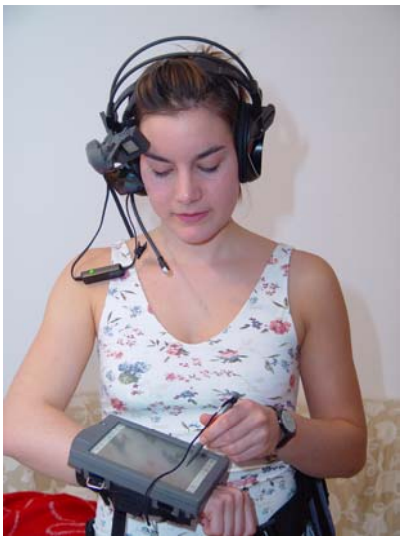


Figure 37: working with the Xybernaut Equipment

on an Xybernaut Atigo T [8] and on a weaker wearable computer (Figure 36, Figure 37) from the same company. The underlying network based on TCP/IP packages routed via an access point was of course wireless.

III.6. Evaluation

The positive experiences I made and the new ideas that came up during the final testing of the SVSDM prototype and during the composition of this document require a critical analysis of it. The prototype has a lot of advantages to offer, most of them because of CORSO but it is still not perfect. If a new version should be justified a lot of improvements need to be made first.

III.6.1. Advantages

- Communication objects in the space are persistent and can be recovered after system or network failures
- Every device is always synchronized by using shared objects. There is no explicit need for a function that informs the others if the device goes on or offline
- At any time each participant has a consistent view of all objects
- CORSO supports long running distributed and heterogeneous transactions
- A monitor can be installed at any place to get informed about changes in the space
- Network traffic is minimized through access to the local SVS cache
- The abstraction the SVS space offers decreases the effort put into the implementation of applications

III.6.2. Suggested Improvements

- The communication protocol between the worker and the master process can be improved. After sending a request to the master process the worker starts in the current version a new and completely independent notification service in order to look for incoming answers. This is not necessary at all. A very easy solution would be the usage of synchronous reading. This means that the worker process tries to read the *WCAC* but

CORSO blocks because the timestamp rule is not satisfied. When the master process finishes the request it writes the result into that *WCAC* which increments the timestamp of the object. A few moments later CORSO does not block the worker process any more, because the rule is not broken. This would allow continuing work with the latest response.

- Currently the *WFPC* object has a variable called *instancefiles* which stores additional data in order to support the agents with more specific information. Theses added files should be saved in an additional object, since the size and number of the files is not known and could increase the size of the object insupportably. The change would result in smaller *WFPC* objects meaning that distribution and propagation of newly added packages and updates of already existing ones can be performed faster.

- CORSO is on the one hand not a database but from transaction's point of view it could be seen as one. However, it is not suggested to store any data, especially big ones, in it. Instead of the files themselves, a url could be provided by the user or the using system and saved in the objects. By the way this would also speed up distribution of the packages. If an agent selects a package, a download manager could be started that takes the provided url and downloads the requested files. The url could base on http, ftp or anything else, but it needs to be understood by both the client and the target system.

- If the worker process would like to fetch a package it has to clone the wished one. The reason is that the client needs the primary copy of the object, if it wants to write into it. This would not be necessary, if the prototype had an answer object offered. In that case it would be enough to have the package for reading in the cache of the local space and for writing just a new answer object. If the worker would like to upload the answer to the global space, it just takes the answer object instead of the package.

- The communication protocol does not support worker initiated inserts into the space. The current version supports only packages that are distributed by the master process. In the next one the worker should be able to upload its own packages as well.

- The profile service has not been implemented completely. From the author's point of view, this is not necessary in the core SVSDM application. Actually there are two possibilities how SVSDM could handle profiles. Either not at all or SVSDM set up rules that define how a profile has to look like. The second case is the most crucial one since it might be possible that not all kinds of authorization policies or customer requirements can be fulfilled. This might be easily possible, if wrong file specifications have been set up. The second case is the easiest one. The only thing the SVSDM has to do is on the one hand to enable the master process to specify profiles which has to be done via files and on the other hand to give them to the worker process. This will then do the matching with the client's profile and informs the SVSDM only, if the entry is authorized to be shown or not.
  It is important that matching is not done by SVSDM. The number of different file types and convention could not be handled at all.

- At the current version of SVSDM, the only type of communication does exist between the master and worker processes only. Actually this is not sufficient enough since agents might be grouped as well into agencies. This implies that a three layer distribution – master, agency and agent - is not supported. The outlined problem here is that distribution between the agencies first and from there between the agents is not supported. A big step towards this aim would be the use of profiles. These would allow running n-layer distribution applications.

- Every package put into the space so far comes from the master process. Enabling distribution of packages from lower levels of the n-layers would make the entire system more flexible and dynamic.

- So far there is only one master process working in the SVSDM prototype. This one keeps the primary copy of each communication object and performs requests coming from the numerous worker processes. In order to make the system more scaleable, reliable and to get rid of this bottleneck, there is a need for a distribution of these copies respectively the requests. Therefore, first of all policies have to be set up that have to be fulfilled by the machines that would like to overtake a part of the work

of the master process. These policies have to require from the machines that they can be up and running for at least 99,999% of the year. Second, a mechanism has to be developed, e.g. a load balancer that distributes requests to those mainframes equally. This can be done by “shifting” the various *WCRC* objects to master processes that are not so busy. Furthermore this application has to have to capability to tell any master process where the primary copy of a communication object is currently located.

III.7. An alternative Approach

CORSO is not the only technology the current SVSDM prototype could have been implemented with.

The use of a database was one kind of technology that could have been taken as well. In that case the database represents the space that keeps the data currently stored in the *WFPC* and *WFC* objects. This DB is installed and running on a single server. This is necessary since the use of a distributed database laid over all worker machines would lead to partial loss of data as long as workers stay offline.

The master process running on the server can insert, delete and update table entries via sql commands. Logging and monitoring of changes can be achieved by using the DB manager.

More difficult is to design the way of communication of the worker process. One way was to use the http protocol. The worker browses through the list of available packages, which has been created according to the worker's profile, and downloads the packages she/he needs. In order to skip network failures a download manager should be used. The files can be saved then in special directories on the local hard disk. Additional parameters, like *TTL SVS* or *TTL worker*, have to be stored in an extra file. After the files have been downloaded, the worker process has to place a request asking the server to mark the package selected. If it receives a positive acknowledgement, the worker can go offline and execute the downloaded files. If the worker finishes work it uploads the answer to the server that inserts it into the DB and marks the entire package as done.

Authentication of the worker can be achieved by requesting username and password and verifying them over a secure connection, like https. All these necessary services, needed to run SVSDM, would require another server, a web-server.

The main question in this section however is how much effort needs to be put into this kind of system compared to the one with CORSO, especially if some of the suggested improvements, mainly focusing on scalability, had to be implemented as well.

In that case improvements in the current system would only require minimal implementation efforts. Starting new master processes on additional servers

that can fulfill the set up policies is nothing to speak of. Reorganizing the *WCCs* objects, to be more exact specifying which server handles which requests, requires minimum of changes in the *WorkerCommunicationThread*.

On the other hand a web and database based system would need more attention. The database had to be modified in a way that allows distributed synchronization of it over multiple servers. The number of web-servers needed to be increased as well. This would also ask for another server that distributes incoming requests to the various web servers.

IV. Conclusion

In this master thesis a distribution pattern called SVSDM based on the CORSO architecture has been presented.
On the one hand it has been shown that it is pretty easy to develop distributed applications by using the CORSO API. Furthermore it gives the developer an abstraction level that relieves him from bothering about topics like location migration, replication, persistence or any kind of failure. Additionally CORSO supports committing actions in a global transactional secure way.

On the other hand it has been described that the SVSDM prototype can be seen as a pattern, making use of these advantages and offering a way how workflows can be distributed to various numbers of clients. Actually the current prototype is based on a specific example. There workflows are packed into objects, so called packages, which are present in the space. These can be grabbed by workers and used for further processing. The result is then written back into the space. Moreover SVSDM provides monitoring and logging facilities that keep everyone informed about anything that happens in the space.

But research on that area is not over yet. As suggested the current prototype can and should be modified in a way so that any kind of information can be distributed. In that case the SVSDM should be seen on the basic level as parcel service, like UPS, with the only task to reliably deliver packages from one place to the other.

On the higher level SVSDM offers the usage of profiles that give the user of the system the capability of specifying who is allowed to fetch a package and who is not. This allows distribution not only between participating clients; it has the power as well to distribute between specific predefined working groups.

References

1. Tanenbaum, Andrew S., Steen, Maarten van 2002, *Distributed Systems Principles and Paradigms,* Prentice Hall, Inc.

2. Tecco AG, *CORSO Version 3.3 Tutorial,* downloaded on the 10th of June 2004 from http://www.complang.tuwien.ac.at/eva/Download/CORSO/

3. Giacomo Cabri, Letizia Leonardi, Franco Zambonelli, *"MARS: a Programmable Coordination Architecture for Mobile Agents"*, IEEE Internet Computing, volume 4, number 4, pages 26-35, year 2000

4. Werner Kuschl, *"Space-Based versus Message-Passing Communication A Comparison"*, downloaded on the 1st of September 2004 from http://webster.fh-hagenberg.at/staff/kurschl/pubs/ TR.2004.01.SpaceBasedvsMessagePassing.pdf

5. Eric Freeman, *"Make room for JavaSpaces"*, downloaded on the 1st of September 2004 from http://www.javaworld.com/javaworld/jw-11-1999/jw-11-jiniology.html

6. IBM, *"Business Process Execution Language for Web Services"*, downloaded on the 1st March, 2004 from http://www-106.ibm.com/developerworks/webservices/demos/ bpelws/

7. *"Entwicklung eines SVS Gateway zur Verteilung von Arbeitsaufträgen"*, Marcus Mor, Master thesis in work

8. Atigo T produced by Xybernaut GesmbH, http://www.xybernaut.com/ Solutions/product/Atigo_T_tech.htm

9. Bernhard Angerer, *"Space-Based Computing"*, downloaded on the 21st of November 2004 from http://www.onjava.com/pub/a/onjava/ 2003/03/19/java_spaces.html

10. eva Kühn, *„Virtual Shared Memory for Distributed Architecture"*, Nova Science Publishers, 2001

11. Fabian Schmied, eva Kühn, *„Distributed Peer-to-Peer Application Development with Declarative and Aspect-Oriented Techniques“*, accepted for publication, 1st International Symposium on Leveraging Applications of Formal Methods (ISOLA 2004), LNCS, Cyprus, 30th October - 2st November 2004

12. Bernhard Angerer, Werner Kuschl, Peter Lieber, *“MAINSTTREAM GRID COMPUTING – Software Entwicklungen zur globalen Vernetzung”*, Objektspektrum 2004, Nr. 6

Abbreviations

BPEL	Business Process Execution Language
BPEL4WS	Business Process Execution Language For Web Services
CONST	Constant communication objects
CORSO	Coordinated Shared Objects
OID	Object Identification
TTL	Time To Lease
VAR	Variable communication objects
VSM	Virtual Shared Memory
WCAC	WorkerCommunicationAnswerContainer
WCC	WorkerCommunicationContainer
WCCs	WorkerCommunicationContainers
WCLC	WorkerCommunicationLogContainer
WCRC	WorkerCommunicationRequestContainer
WFC	WorkFlowContainer
WFPC	WorkFlowPackageContainer
WSDL	Web Service Description Language

Figure List

Appendix

API WorkFlowPackageContainer

WorkFlowPackageContainer

collapsed

- WorkFlowPackageContainer (ID:CorsoVarOid, TTLContainer:LongInteger, TTLWorker:LongInteger, instancename :String, instancefiles :Byte[], workflowref:CorsoVarOid, workflowname :String)
- WorkFlowPackageContainer (oid:CorsoVarOid, tx:CorsoTopTransaction)
- clone (localOid:CorsoVarOid) : Object
- displayFullLn () : String
- getAnswerFiles () : Byte[]
- getAnswerFilesNumber () : Integer
- getAnswerTitle () : String
- getAnswered () : Boolean
- getAnsweredInString () : String
- getBeginDateQueue () : Date
- getBeginWorker () : LongInteger
- getBeginWorkerDate () : Date
- getBeginWorkerInString () : String
- getEndQueue () : LongInteger
- getEndQueueDate () : Date
- getEndWorker () : Date
- getEndWorkerInString () : String
- getID () : CorsoVarOid
- getInstanceFiles () : Byte[]
- getInstanceFilesNumber () : Integer
- getInstanceName () : String
- getLocalID () : CorsoVarOid
- getRowInputAdmin () : String[]
- getRowInputWorkerLocal () : String[]
- getRowInputWorkerServer () : String[]
- getSelected () : Boolean
- getSelectedBy () : String
- getSelectedInString () : String
- getSelectedValue (columns :String[]) : String[]
- getStatus () : String
- getTTLContainer () : LongInteger
- getTTLWorker () : LongInteger
- getValid () : Boolean
- getValidInString () : String
- getWorkFlowName () : String
- getWorkFlowRef () : CorsoVarOid
- isAnswered () : Boolean
- isExpiredCorso () : Boolean
- isExpiredWorker () : Boolean
- isForeverInSVS () : Boolean
- isSelected () : Boolean
- isValid () : Boolean
- read (data:CorsoData) : Void
- setAnswer (Title:String, ansfiles :Byte[]) : Void
- setBeginDatequeue (newbegin :Date) : Void
- setBeginWorker (newtime :Date) : Void
- setInstanceFiles (newdata:Byte[]) : Void
- setInstanceName (newname :String) : Void
- setReleaseSelected () : Boolean
- setSelectedBy (user:String) : Void
- setStatus (newstatus :String) : Void
- setTTLContainer (newttlcontainer :LongInteger) : Void
- setTTLWorker (newttlworker:LongInteger) : Void
- setValid (valid:Boolean) : Void
- setWorkFlowRef (newref:CorsoVarOid, name:String) : Void
- timeToExpireCorso () : Date
- timeToExpireWorker () : Date
- write (data:CorsoData) : Void

API ModificationOfNotification

«final»

ModificationOfNotification

collapsed

- ModificationOfNotification (site:String, port:Integer, domain:String, user:String, password:String, label:JLabel)
- addWindow (window:Object) : Void
- completeUpdateOfCorsoConnection () : Void
- disConnect () : Void
- establishCorsoConnection (createOids:Boolean) : Void
- getConnectionDomain () : String
- getConnectionPassword () : String
- getConnectionPort () : Integer
- getConnectionSite () : String
- getConnectionStatusLabel () : String
- getConnectionStrategy () : CorsoStrategy
- getConnectionUser () : String
- getContainerOidCOM () : CorsoVarOid
- getContainerOidCOMA () : CorsoVarOid
- getContainerOidCOMR () : CorsoVarOid
- getContainerOidWF () : CorsoVarOid
- getContainerOidWFP () : CorsoVarOid
- getCorsoConnection () : CorsoConnection
- getRootCOMA () : CorsoNotification
- getRootCOMA (tx:CorsoTransaction) : CorsoNotification
- getRootCOMR () : CorsoNotification
- getRootCOMR (tx:CorsoTransaction) : CorsoNotification
- getRootWF () : CorsoNotification
- getRootWF (tx:CorsoTransaction) : CorsoNotification
- getRootWFP () : CorsoNotification
- getRootWFP (tx:CorsoTransaction) : CorsoNotification
- getTellingOid () : CorsoVarOid
- getTopTrans () : CorsoTopTransaction
- getWCCNames () : String[]
- getWCCs () : Vector
- getWCLC (uname:String) : WorkerCommunicationLogContainer
- getWindows () : Vector
- isConnected () : Boolean
- isExistingWCC (username:String) : Boolean
- isExistingWF (ID:String, oidsearch:Boolean) : Boolean
- isExistingWF (searchOid:CorsoVarOid) : Boolean
- isExistingWFP (ID:String, oidsearch:Boolean) : WorkFlowPackageContainer
- isExistingWFP (WFP_Name:String, WF_Name:String) : WorkFlowPackageContainer
- isExistingWFP (existsOid:CorsoVarOid) : Boolean
- openFile (answerfiles:Byte[]) : Void
- placeNewWCC (name:String) : Void
- placeNewWF (WorkFlowTitle:String, mainfiles:Byte[], additionalfiles:Byte[]) : WorkFlowContainer
- placeNewWF (WorkFlowTitle:String, mainfiles:Byte[], additionalfiles:Byte[], tx:CorsoTopTransaction) : WorkFlowContainer
- placeNewWFP (TTLContainer:LongInteger, TTLWorker:LongInteger, InstanceTitle:String, instancefiles:Byte[], referworkflow:CorsoVarOid, referworkflowname:String) : WorkFlowPackageContainer
- placeNewWFP (TTLContainer:LongInteger, TTLWorker:LongInteger, InstanceTitle:String, instancefiles:Byte[], referworkflow:CorsoVarOid, referworkflowname:String, tx:CorsoTopTransaction) : WorkFlowPackageContainer
- removeReferenceFromWF (wfoid:CorsoVarOid, RefName:String, tx:CorsoTopTransaction) : Boolean
- removeWCA (oid:CorsoVarOid, top:CorsoTopTransaction) : Boolean
- removeWCR (oid:CorsoVarOid, top:CorsoTopTransaction) : Boolean
- removeWF (delOid:CorsoVarOid) : Void
- removeWF (delOid:CorsoVarOid, tx:CorsoTopTransaction) : Void
- removeWFP (delOid:CorsoVarOid) : Void
- removeWFP (delOid:CorsoVarOid, tx:CorsoTopTransaction) : Void
- removeWindow (window:Object) : Void
- saveFile (ID:CorsoVarOid) : Void
- setConnectionDomain (domain:String) : Void
- setConnectionPassword (password:String) : Void
- setConnectionPort (port:Integer) : Void
- setConnectionSite (site:String) : Void
- setConnectionStatus () : Void
- setConnectionUser (user:String) : Void
- setConnectionVariables (site:String, port:Integer, domain:String, user:String, password:String) : Void
- setContainerOidForCOM (createOid:Boolean) : Void
- setContainerOidForCOMA (createOid:Boolean) : Void
- setContainerOidForCOMR (createOid:Boolean) : Void
- setContainerOidForWF (createOid:Boolean) : Void
- setContainerOidForWFP (createOid:Boolean) : Void
- setContainerOids (createOid:Boolean) : Void
- setCorsoConnection () : Void
- setStrategy () : Void
- setTellingOid (createdOid:Boolean) : Void
- storeFile (answerfiles:Byte[], path:String) : Void
- updateWF (struct:WorkFlowContainer, mainfiles:Byte[], additionalfiles:Byte[]) : Void
- updateWF (struct:WorkFlowContainer, mainfiles:Byte[], additionalfiles:Byte[], tx:CorsoTopTransaction) : Void
- updateWFP (struct:WorkFlowPackageContainer, TTLContainer:LongInteger, TTLWorker:LongInteger, instancefiles:Byte[], referworkflow:CorsoVarOid, referworkflowname:String) : Void
- updateWFP (struct:WorkFlowPackageContainer, TTLContainer:LongInteger, TTLWorker:LongInteger, instancefiles:Byte[], referworkflow:CorsoVarOid, referworkflowname:String, tx:CorsoTopTransaction) : Void
- writeToTellingItem (tx:CorsoTopTransaction) : Void

API ModificationOfNotificationLocal

ModificationOfNotificationLocal

collapsed

- ModificationOfNotificationLocal (site:String, port:Integer, domain:String, user:String, password:String, contnameWFP:String, contnameWF:String, ServerIP:String, labelworker:JLabel)
- ShowHistory () : Void
- completeUpdateOfCorsoConnection () : Void
- disConnect () : Void
- establishCorsoConnection (createOid:Boolean) : Void
- executeGUI (loccont:WorkFlowPackageContainer) : Void
- getConnectionDomain () : String
- getConnectionPassword () : String
- getConnectionPort () : Integer
- getConnectionSite () : String
- getConnectionStatus () : Boolean
- getConnectionStrategy () : CorsoStrategy
- getConnectionUser () : String
- getContNameSite () : String
- getContNameWF () : String
- getContNameWFP () : String
- getContainerOidWF () : CorsoVarOid
- getContainerOidWFP () : CorsoVarOid
- getCorsoConnection () : CorsoConnection
- getMyLogName () : String
- getMyLogOid () : CorsoVarOid
- getRootWF () : CorsoNotification
- getRootWF (tx:CorsoTransaction) : CorsoNotification
- getRootWFP () : CorsoNotification
- getRootWFP (tx:CorsoTransaction) : CorsoNotification
- getServerSite () : String
- getTopTrans () : CorsoTopTransaction
- goToBed (time:LongInteger) : Void
- isConnected () : Boolean
- isExistingWFP (cont:WorkFlowPackageContainer) : Boolean
- moveFromLocaLToServer (loccont:WorkFlowPackageContainer , reqcont:WorkerCommunicationRequestContainer) : Void
- moveFromServerToLocal (wfpcont:WorkFlowPackageContainer , reqcont:WorkerCommunicationRequestContainer) : Void
- removeWorkFlow (ID:String, oidsearch:Boolean) : Void
- sendRequestAndConfirm (reqcont:WorkerCommunicationRequestContainer , top:CorsoTopTransaction) : Void
- setConnectionDomain (domain:String) : Void
- setConnectionPassword (password:String) : Void
- setConnectionPort (port:Integer) : Void
- setConnectionSite (site:String) : Void
- setConnectionStatus () : Void
- setConnectionUser (user:String) : Void
- setConnectionVariables (site:String, port:Integer, domain:String, user:String, password:String, contnameWFP:String, contnameWF:String, serverIP:String) : Void
- setContNameSite (con:String) : Void
- setContNameWF (contnameWF:String) : Void
- setContNameWFP (contnameWFP:String) : Void
- setContainerForMyLog (created:Boolean) : Void
- setContainerOidForWF (create:Boolean) : Void
- setContainerOidForWFP (create:Boolean) : Void
- setContainerOids (create:Boolean) : Void
- setCorsoConnection () : Void
- setMyLogName () : Void
- setServerSite (serverSite:String) : Void
- setStrategy () : Void
- transferToBPEL4J (loccont:WorkFlowPackageContainer) : Void
- updateLogAtServer (reqcont:WorkerCommunicationRequestContainer) : Void
- writeIntoMyLog (CODE:Integer, msg:String) : Boolean