8

Service prototyping in the OVOPS environment

Panu Puro[a], Jarkko Sonninen[b]
[a]Lappeenranta University of Technology,
Data Communication Laboratory,
P.O.Box 20, FIN-53851 Lappeenranta, Finland
Tel. +358 53 574 3613, Fax. +358 53 574 3650
[b]Systems Software Partners, Laserkatu 6,
FIN-53850 Lappeenranta, Finland
Tel. +358 53 574 3612, Fax. +358 53 574 3650

Abstract

This paper will describe the OVOPS environment. The OVOPS environment is being developed by Telecom Finland and Lappeenranta University of Technology. It supports the design, implementation and debugging of distributed applications. Programs can be implemented operating system independently so that applications developed or embedded in the OVOPS environment can be ported to any system supporting OVOPS.

OVOPS can be used as a platform for implementing intelligent network services. The idea of using OVOPS in service creation is that it provides a way to design and implement services in an object-oriented and modular way. It encourages implementing software in modules (tasks) and provides facilities for message-based programs. Provided facilities are, for example, a communication channel between tasks with messages, scheduling of tasks, a base class for a task, other useful classes and a user interface for debugging.

Service creation is considered from a point of view of a programmer. OVOPS itself can be seen as a bottom-up approach for implementing message-based programs like intelligent network services. The bottom-up approach is good for fast prototyping and with it we can see the actual needs and problems of a service easily. A programmer may also affect the service implementation to make it more effective, which is usually impossible when high-level tools are used.

J. Harju et al. (eds.), *Intelligent Networks*

INTRODUCTION

Since the term "intelligent network" has been introduced we have been talking about fast service creation. It is true that with flexible architecture and network elements it is possible to offer all kinds of services without modifications to hardware. This is fundamental if we want to continue to develop intelligent networks.

Before any company producing intelligent network services can say to have a fast service creation cycle there is a lot work do to. Some work must be done because you cannot buy a ready system suitable for your needs. This can be said to be the situation at the moment but it is just matter of time when reasonable systems appear due to huge market in intelligent network business.

For the need of software tools the development of OVOPS was started in the summer of 1993. The goal was to implement an environment for event-based applications and especially for protocols. OVOPS consists of the OVOPS library and user interface. The user interface provides features to debug tasks in the OVOPS system, which is essential help in the development phase of software.

In the future we are planning to research the possibility to extend the OVOPS environment for intelligent network creating an environment for testing and developing intelligent network services.

SERVICE CREATION

Service creation can be seen as software engineering in a special environment, intelligent network. It has many problems to be solved before all benefits are exploited. Problems are raised by group work as well as other usual issues in software engineering.

An idea behind service creation is to produce services for intelligent network architecture. However, we are not just producing services but we want to produce them using as little effort as possible. This is an ideal goal that needs planning to be successful. Software tools are often introduced when we are talking about efficient software engineering although other possibilities exist as well.

A good way to minimize the time used in software development is to reuse code. This is one of the goals of OVOPS. The OVOPS library contains classes that can be used in all protocol implementations, which reduces the implementation time and number of programming errors. The needed code is in the OVOPS library that is ready and heavily used in other programs.

To produce a good piece of reusable software is quite difficult because it should be as general as possible or it should be easy to modify for different needs. Usually reusable software is not done because it takes extra effort and planning. This is effective but it could be more profitable in the long run to use this extra effort. Otherwise, the wheel is reinvented many times.

Software libraries are logical places to store general and useful software that should be available for every one in your organization. For example in an intelligent network service one library could be a group of functions for accessing a database. In this way you have a standardized way to do it and all of your services may use this same interface, which is a great help when you want to use another (different) database system for your services.

Unfortunately, using libraries is not so simple when managing large software projects. Some guidelines should be given for how to correct errors in the library and who will test the library. Addition to that every library should have a librarian who is responsible for maintaining the status of the library. The librarian is the person who answers all questions asked about the library by users.

Having useful libraries increases possibilities to prototype services because you have ready and useful code ready and available. Libraries have one good feature. Using libraries reduce the amount of code that is copied. This is good because if the copied code contains an error, it must be corrected only in one place increasing the quality and consistency of software and decreasing the amount of code needing maintenance.

All these issues mentioned above should be guided for programmers. Guideline helps to achieve all benefits of good software engineering.

After all, software tools are essential for productive software engineering. A tool should be flexible enough providing a possibility to employ it as you wish. It is too typical that the tool is not designed to work with other tools. Flexible tools, otherwise, are often complicated to use making it hard to get familiar with them.

OVOPS

Idea

OVOPS is designed to be a platform for other software tools. It is flexible enough with only few restrictions. One of the main benefits of OVOPS is that it provides software library that contains useful classes often needed in your software. Programming in the OVOPS environment modular architecture is encouraged improving compatibility and interoperability of your programs.

Shortly the idea behind OVOPS is to provide:

- Scheduling of tasks
- Message passing method between tasks
- An interface to devices (the operating system)
- User interface for debugging
- Timers, frame and other useful classes

OVOPS is meant to be suitable for different areas. This means that the tool must be flexible. OVOPS does not restrict the structure of the program. The system may consist of as many tasks and schedulers as wanted. The number of the channels between tasks can be specified by the user.

Overview

OVOPS is a practical approach for developing event-based applications. It supports the design, implementation and testing of event-based applications by providing classes for tasks, messages, schedulers, ports and devices. These classes are operating system independent and communicate by given message structures so that applications developed or embedded in the OVOPS environment can be ported to any system supporting OVOPS.

OVOPS is an environment where different applications, databases and protocols can be implemented and interfaced. There are no formal restrictions from the OVOPS side to the specifications and languages used in the interfaced applications. Foreign environments can be supported by specifying an embedding OVOPS task class for them.

In the OVOPS design performance was one of the most important aspects. So, the OVOPS library provides basic objects, or services. Of course, the services of the OVOPS library can be enriched by adding or writing specialized libraries.

OVOPS is suitable for being an environment in which tools and formal description techniques can be used. OVOPS can be said to be a practical approach to problems in large distributed applications and it should be used as a platform for software engineering.

An OVOPS system consists of tasks and channels. The channel connects two tasks to each other. Tasks communicate through a channel by sending messages. The message passing is asynchronous.

Tasks in OVOPS are independent from other tasks. The only thing a task knows about another task is an interface. The interface consists of a port and messages that can be send through the channel. A developed OVOPS system can be divided into several processes that can run in different computer systems.

The operation of tasks is event driven. All tasks have to perform their work gradually because the system is not pre-emptive. Only one task is executed at the same time. The task must release the control voluntarily. If not, all the other tasks inside the same process will be waiting the end of the execution of the running task.

The executions of tasks inside each process are scheduled by a user written scheduler, or a scheduler provided by the OVOPS library. Schedulers can be scheduled by other schedulers. So, it is possible to build a complex hierarchy of tasks. The top level scheduler is executed by the main loop of the process. There is no so called kernel in the OVOPS system but the functionality of OVOPS is based on contracts between classes.

Complex OVOPS systems with multiple schedulers have to be designed carefully and formal tools to prove and simulate complex behavior are needed if you want to be absolutely sure that you get benefits using a complex structure.

Usually, all processes need access to devices, a terminal for example. Physical devices are handled by driver tasks. The driver task is a task that uses a device class as an interface to the physical device and forwards data from or to the device by sending messages. The driver task is needed due to the nature of the system and the tasks are wanted to be kept independent from the operating system used.

CONCEPTS

VOPS

At first we could consider what is a VOPS (virtual operations system or virtual operating system). Why do we need the VOPS? The VOPS can be seen as a layer above the operating system. It helps a programmer to implement more modular programs, which are usually event-based applications, that communicate by using messages.

With the term VOPS we understand the platform that gives additional help and services for the programmer to implement event-based applications. OVOPS provides services for

debugging and tracing system events, scheduling of tasks and transferring messages. These are same kind of services that the operating system usually provides but these are intended to be independent from the operating system improving portability between different operating systems. No assumptions are made if the operating system provides help for multi-tasking or not, which can be very true for embedded systems. The VOPS style architecture may sometimes be more efficient than the application that deploys multiple processes and interprocess communications for the same purpose because communication is lighter and there is no need for context switches, for instance.

OVOPS System

With an OVOPS system we mean the group of user written tasks and the system tasks that the OVOPS provides. It is the entire system that performs its work, for example, a protocol stack.

The term is later used to express the OVOPS library and the user interface, in some cases.

OVOPS Class Library

The purpose of the OVOPS class library is to provide standard services for applications implemented with OVOPS. The library is not intended to cover all the needs that a programmer has but it provides some fundamental services needed in event-based applications.

The library cannot support all needs in different software areas. So, all additional services are supported by specialized libraries or tools for a particular purpose. In this way the OVOPS library can be kept as standard and stable as possible.

User Interface

OVOPS can be divided to two parts, the OVOPS library and the user interface module. The user interface provides help for tracing and debugging of user defined and implemented tasks. The main purpose of the user interface is to provide help for tracing messages, i.e. events happened, in the system. This kind of help lacks from the debuggers that are also very useful in programming beside the user interface. That means that the user interface is in principle only used when you are developing an OVOPS system consisting of OVOPS tasks and channels interconnecting them. The user interface is an important part of the OVOPS system although it is used only in a development phase.

The user interface can be said to be some kind of debugger because it provides same services and features. However, the user interface needs help from the user because it does not use any debugging information on object files but it needs hooks to the user tasks. Hooks are C preprocessor macros, which are provided by the user interface and contain function calls that inform the user interface about variables in the user task. The hooks are for getting information and modifying variables in the tasks by using the user interface. Only hooked variables can be seen and set by the user interface.

There are no hooks in the code in the OVOPS library, which means that the classes of the OVOPS library are the exact code that will be in the developed product without any preprocessor's macros. In other words the OVOPS library can be linked with the development phase code and the product code. However, the hooks in the user written tasks should be able to be removed when the user interface is no longer wanted.

Any printing and setting routines are not wanted to implement to the classes of the OVOPS library, i.e. the code needed only in a development phase, for example print functions, is kept separately. This is the reason why hooks are needed. The actual printing and setting routines, which have access through hooks, are in the user interface, not in classes in the OVOPS library.

Tasks

Tasks are executable classes in the OVOPS. Because the system is not pre-emptive tasks have to perform all processing gradually. In other words a task has to release the control to the other tasks. Otherwise, the other tasks are and will be waiting execution forever, and only one task is running.

It is natural to model a task by using a finite state machine because tasks are asynchronous and event-driven. The task usually has certain messages that are input events and they produce transitions and output messages when processed. This model suits well for the nature of tasks in OVOPS. However, OVOPS itself does not provide any help for modeling the finite state machine. It is supposed to be provided by other tools.

Tasks do not have any predefined structure, except some variables needed by the system and message queues. The user's job is to build interfaces, any number of interfaces, to the task and to divide the application into modules, or tasks, and, of course, to write the functionality of the task to the run function. The run function is the function that is called by the scheduler when the task is decided to be executed.

In OVOPS there are two kinds of tasks, Otasks and EventTasks. The Otask is the base class of all executed tasks. The EventTask is a task that has a message queue and it is usually inherited by the user written tasks. EventTasks have predefined run function that takes a message from the message queue, if present, and calls an execute function with the message. The execute function should have a user-written body. Only special kinds of tasks do not need message queues, i.e. they are inherited from the Otask directly. These are, for example, system tasks, schedulers, I/O handler and timer task.

Message Passing

The message passing is one of the most important features in OVOPS. It is handled by port classes. The user makes instances, as many as needed, of the port and connects them to other ports. One port can be connected to only one port at the same time. In other words there are not predefined ports in any task.

The port has to know to which task it belongs because it has to be able to store the received message to the message queue of the task.

The OVOPS library contains two different ports. One is just for use of inside the OVOPS system in the operating system process. The other is supposed to be used for communication between two or more operating system processes. It uses UNIX or Internet sockets as a transport service. The user may use these two ports as base classes and build specialized (function) interface for messages sent through the port. This is one way to simplify and to hide the sending of messages improving the readability of the program.

All message passing in OVOPS is asynchronous. A task never blocks waiting on the actions of the recipient to which it sends a message. This is a part of the nature of the system, no task

can block unless it is in its own process. Asynchronous message passing makes it possible to implement efficient tasks, but it makes it also much harder than doing it in an easy way. With a complex code that models a finite state machine you are able to handle many events concurrently, instead of handling events in particular order sequently without many concurrent contexts.

Ports are designed as light as possible making the over-head of message passing low, but of course there is always a little over-head when sending messages. You should be careful when giving parameters to messages trying not to copy parameters, which may easily low down the performance of your system. Large data structures as parameters should always be passed by using pointers and references.

In practice almost all the messages are inherited from the message class of the OVOPS library because it does not contain any members that can store user's data. The OVOPS message contains just port addresses, the ports that the message has gone through.

Distribution

By the term distribution we mean in the OVOPS world how to connect tasks in different operating system processes. As described before the message passing is provided by ports and by using socket ports data can be transferred between two processes. UNIX or Internet domain sockets can be used.

Unfortunately communication between two processes is not so easy. The programmer must provide her own encoding and decoding functions that take care of transforming the message to a stream of bits for all the messages sent through the socket ports. Of course, you can use existing coding tools. But anyway, there is some over-head transferring messages to another process, or computer, both in development phase and run-time. In development phase you should provide the coding functions somehow and in run-time the messages have to be coded and decoded before and after sending through the socket.

INTEROPERATION OF CLASSES

OVOPS does not have so called kernel because all the classes are taken in use by the user. The user has to even construct the main loop of an OVOPS system for correct operation by herself. This all means that the operation of the system is based on contracts between classes of the OVOPS library. The following text describes these contracts and the functionality of classes.

Of course, if you are writing your own classes by inhering from classes of the OVOPS library you can change functionality, but you have to be sure that your classes will work with existing classes. Normally, only relationship between a port and event task is interesting for the user that implements software using OVOPS. Other relationships are useful for OVOPS developers, persons who write their own schedulers, for example. However, sometimes detailed information helps to understand the system better.

Port

Between two ports are only few contracts that are only used when the port is the basic port (Port). Your own modified ports may operate differently, but the interface of ports remains the same.

When a message is sent by calling putMessage function the port checks if the port is connected, that is the port has a reference to another port to which it may want to send messages. If the port is connected the port calls the getMessage function of the port of the other end and causing it to receive the message. A message can be transferred only if the port is connected to another port. The getMessage stores or handles the message, or whatever. We will describe this later.

Because ports have references to other ports they have to communicate with each other when a port is disconnected. This confirms that a port does not have a reference to another port that does not exist.

Ports can be disconnected by calling disconnect member function, or ports are disconnected automatically when they are deleted. The disconnect function clears the reference. So, the port is no longer connected.

When a port is connected or disconnected a short handshaking is performed. When the connect is issued the port calls the connectRequest function of the port of the called end that calls the connect function of the task. After this the connect function of the task of the caller end is called then the connection is ready. If the connection was refused the disconnectRequest is issued to the port of the called end.

And when disconnecting the disconnect function of the task is called first. Then the disconnectRequest is issued to the port of the called end and the connection is down. The disconnectRequest function will call the disconnect function of the task.

EventTask and Port

A port has to belong to an event task, which is a task with a message queue, if the default port class is used. Otherwise, the port cannot do anything with the message that was received. The task must be known because the port calls the save function of the event task to store the message into a message queue to be handled later.

There are also two other functions in event tasks that ports call. When a task connects a port the connect function of the task is called in both ends. In both ends it can be used to monitor connections or it can be used for accepting or refusing connections.

Almost the same situation is when a port is disconnected. The disconnect function of the task is called, but it cannot deny the disconnection. The connection is always disconnected, anyway.

Scheduler and IoHandler

An I/O handler is a task but it has some special functions. That is why the scheduler has to know which task is the I/O handler. The special function that the scheduler has to know is the block function. It is called when the scheduler asks the I/O handler to block until something is happened in devices, data can be read or written or a time-out has occurred. The scheduler can call the block function only when there are no messages waiting in any message queue in any

task, that is the load of all tasks is zero. Normally when there are messages to be handled in tasks the scheduler calls the run function of the I/O handler to process I/O requests of devices.

Nothing can be said about the calling frequency of the I/O handler. The current scheduler of the OVOPS library calls the I/O handler when it finishes executing tasks with certain priority. This makes the frequency quite undeterministic because the number of executed tasks varies. The I/O handler is not called before every task is executed because it is not usually needed, of course it depends on how much you communicate with the outer world. Undoubtedly the frequency affects to the performance of the system. If it is called too frequently it is extra overhead and if it is called too rarely the throughput of the system may suffer.

Scheduler and Otask

Every task has the knowledge of its scheduler if it is wanted to be executed. When the scheduler is decided to execute a task it calls the run function of the task. The run function is assumed to the actual task's job.

The task tells the scheduler when it should be executed by calling the request function of the scheduler. It usually happens when the task receives a message and the request function is called by the save function of the task. The scheduler is responsible to take care of executing the requested task as long as the load of that task reaches zero, that is the task has nothing more to do. This minimizes the need of calling the request function repeatedly. We do not assume that every task has a message queue and that is why the Otask class does not have it and we are talking about the load of a task.

When a task is created the constructor of the task calls the inform function of the scheduler to tell that a new task has been created and it should be scheduled.

Analogically, when a task is deleted the destructor of the task tells the scheduler by calling the forget function of the scheduler that the task does not need scheduling.

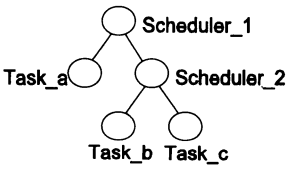

Figure 1. Multiple schedulers. One example of a very simple system structure.

The scheduler itself acts as a task, which makes it possible to a scheduler to have tasks and schedulers to be executed. It should be clear that the use of multiple schedulers should be researched carefully because scheduling takes a certain amount of overhead and the execution of the system becomes easily heavier than expected. But it can also be a great benefit if you are developing a very specialized scheduler for a particular purpose. As you may notice you do not need necessarily the top most scheduler but you can call the second level schedulers yourself reducing the number of schedulers in the system.

The scheduler performs its work gradually, executing only one task at the same run function call. And of course, the run function must not block because all the other tasks are waiting for execution.

IoHandler and Device

The relationship between an I/O handler and devices is much like the relationship between the scheduler and tasks.

A device has to know the I/O handler to which it belongs because it has to tell the I/O handler when it is created and deleted. This is done by calling the inform and forget functions of the I/O handler and the reason is that the I/O handler has to know all the devices in the system because it is monitoring them.

The device changes the status due to functions called by the user. The I/O handler monitors the status and checks if it is possible. For example, when the user requests data to be read the device changes the status for reading. Later the I/O handler checks if it is possible to read the file descriptor associated with the device. If it is possible the specific callback routine is called to handle the actual reading.

INTRODUCTION TO USER INTERFACE

OVOPS User Interface is a separate library. In the core OVOPS there are no support or dependencies for debugging. User Interface Library can be seen as an application or utility library that uses OVOPS.

The final version of the user program is usually meant to use only the core OVOPS library. In debugging phase the programmer could use the user interface library with certain C preprocessor macros. In that way it may be easier to disable the user interface afterwards.

Since there is no debugging information or functions in core OVOPS classes, user must explicitly give to the user interface a reference to her objects. With this technique, the user interface gets a picture of the user system, and can manage it.

In practice the user is mostly interested in running her application and getting information about its status. The user usually initiates system by sending messages through interfaces. Messages start traveling between tasks and system changes its state. The user interface library helps keeping track of what happens by displaying information about tasks and the message before and after its execution.

Commands

The OVOPS user interface is character based. User is prompted for commmand words that are interpreted by the user interface library.

```
Welcome to OVOPS (Object Virtual OPerations System)
Copyright (C) 1993 Telecom Finland

( log macro system Exit alias unalias shell ? )
```

In the topmost level of command hierarchy there are the following commands.

- Macro
- System
- Log
- Alias and Unalias

- Other

At the startup of system a special file ovops.rc is read, and commands in it are executed, just as user would have written them. With this file user can preset variables, aliases and so on to make debugging session more comfortable.

In most cases commands can be abbreviated simply by giving first characters of word. The command that first matches to given word is executed. No ambiguities are tested. OVOPS user interface will give you list of possibilities, if you don't give enough input, or it does not accept your data. You may quote longer texts containing special characters and blanks using single or double quotation marks around text.

Example Session

Here is a short piece of session using OVOPS interface. In the session, first the structure of system is displayed, then the variables of task "a" is looked at, and finally a message is send through upper interface of task "a."

```
net/project/ovops-1.0.1/ui/test:1>./amain

Welcome to OVOPS
Copyright (C) 1994 Telecom Finland

( log macro system Exit alias unalias shell ? )
sys str


Task 'ui' = {
    Port 'output' -> Unconnected
}
Task 'a' = {
    Port 'down' -> c:up1
    Port 'up' -> ui:output
}
Task 'b' = {
    Port 'down' -> c:up2
    Port 'up' -> ui:output
}

Task 'c' = {
    Port 'up1' -> a:down
    Port 'up2' -> b:down
}

( log macro system Exit alias unalias shell ? )
sys a print


Task 'a' = {
    Var 'a' = 0
    Frame 'b' [0]  =  {
        31 31 12
    }
    Timer 't' = 10.000000
    String 'name' = "Task A"
```

```
    Text 'hmm'
    MessageQueue: 0  SystemQueue: 0  Load: 0
    Trace: Message
}

( log macro system Exit alias unalias shell ? )
sys a up

( CrMsg CcMsg DtMsg DrMsg show set print trace struct ? . )

CrMsg CR 123

*** Message to a:up from Outer Space ***
Message 'msg_1' = {
    Enum 'typ' = CR
    Var 'qos' = 123 (1918981664)
}

Got Message: CR
*** Message to c:up1 from a:down ***

Message 'msg_1' = {
  Enum 'typ' = CR
  Var 'qos' = 123
}
```

SIMULATION SUPPORT IN OVOPS

Intelligent Network services are complex program entities and it is difficult to analyze them in other ways than simulation. Simulation provides also a possibility to test various configurations and values of parameters easily.

As OVOPS is a generic programming tool it is relatively easy to build simulation support above it. The OVOPS way to structure functionality as communicating tasks fits well for modeling simulated systems.

Use of simulation

Simulation of IN services can be performed in many levels of abstraction. User can construct an OVOPS system for modeling interactions between network elements such as SCP, SSP and SDP for analyzing performance issues of IN architecture. Such a system could for example give information about overall performance of telecommunication system. There could be certain delays in between entities and amount of traffic may vary.

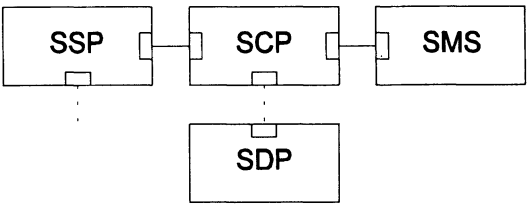

Figure 2. Example simulation system for investigating intelligent network architecture.

Simulation may help also the actual implementation of service. After the service and its environment are specified, a simulation environment for prototype can be used to test the first version. In this kind of environment simulated resources such as databases and SSPs can be used. It is faster to make prototypes without needing to use the real environment, for example telephone exchanges. This applies also for the first testing of service, though the final testing must always happen at the real system.

The interface between the service and execution environment can be kept similar in the simulation environment and real environment, so in principle there are no changes between the simulation version and final version of the service.

The OVOPS system is based on modules, so it is possible to gradually replace simulation modules with real service environment modules and to test at every phase.

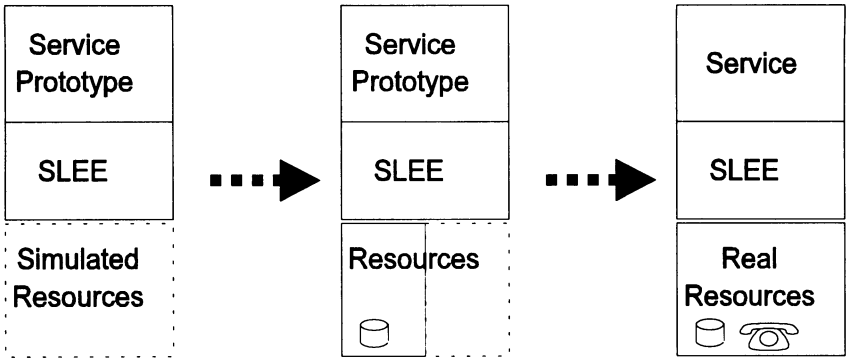

Figure 3. Modularity of OVOPS environment gives a possibility to replace gradually simulation modules with real modules.

OVOPS Simulation Environment

A support for virtual time and scheduling belongs to the OVOPS simulation library. Another important part is statistical functions.

For the IN services the code for simulated resources is also needed. SLEE has to be customized to be useful in simulation. Since the SSP part of service is provided by simulation it is easier to generate service traffic by traffic generator modules.

Logging events for further study and comparison are an obvious and essential feature and support for that is part of the core OVOPS user interface. Ability to control and display statistics is also needed.

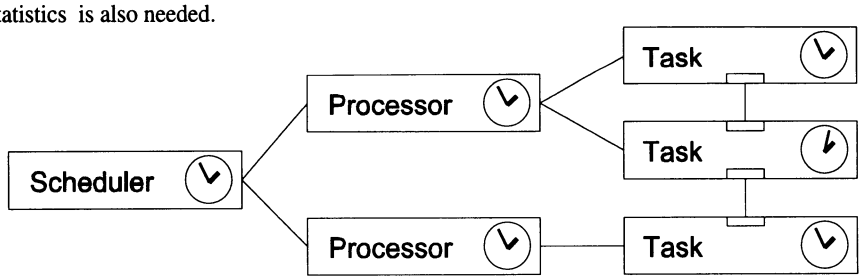

Figure 4. Example simulation system hierarchy

Basic concept in simulation environments is often an *event*, which is an input appearing at a certain moment. The event is mapped in quite a straightforward way to OVOPS Message class with a time stamp.

Usually the OVOPS system consists of tasks connected together with ports and schedulers running tasks. For virtual time, a new concept called Processor has to be introduced. New, modified Task and Scheduler are also required.

Processor models a real world process or a processor. It can be thought as one kind of scheduler that can execute one task a time. The main level scheduler controls processors to execute events in their time stamp order.

Virtual time has to be added to the main OVOPS classes. That is for keeping track which event has to be executed next and to update statistic variables.

The main functionality of the system is implemented in tasks. There are no major logical changes to the tasks in the simulation environment compared to the real environment, since execution and scheduling in a virtual time system matches to the real situation.

The system described in this chapter should fasten the development of service prototypes, aid and speed up testing in the way from prototypes to the final product.

BIBLIOGRAPHY

/1/ Margaret A. Ellis, Bjarne Stroustrup: The Annotated C++ Reference Manual, New Jersey, Addison-Wesley, 1990, 447 p., ISBN 0-201-51459-1.

/2/ CCITT Recommendation I.312/Q.1201 - Principles of Intelligent Network Architecture.

/3/ CCITT Recommendation X.701 - Systems Management Overview (ISO 10040).

/4/ CCITT Recommendation X.722 - Guidelines for the Definition of Managed Objects (ISO 10165-4)

/5/ Tapani Karttunen, Intelligent Network Service Creation Process, Workshop on Intelligent Networks - Proceedings, Lappeenranta, Lappeenranta University of Technology, 1993, 8 p.

/6/ Olli Martikainen, Valeri Naoumov, Konstantin Samouylov, Portable Intelligent Network Software Implementation, Network Information Processing Systems - Proceedings, Sofia, 1993, 6 p.

/7/ OTSO user's guide, Espoo, VTT/TEL, 1992, 132 p.

/8/ Object Virtual OPerations System Manual 1.0, Lappeenranta, Lappeenranta University of Technology, Telecom Finland, 1994, URL: http://shagrat.it.lut.fi/ovops/