TOVE Project
1996 Deliverables

Olli Martikainen
Vesa-Matti Puro
Juhana Räsänen
Timo Pärnänen
Pasi Nummisalo
Petteri Koponen

TABLE OF CONTENTS

1.

ABBREVIATIONS

ATM	Asynchronous Transfer Mode
B-ISDN	Broadband Integrated Services Digital Network
B-ISUP	B-ISDN User Part
CC	Call Control
CPCS	Common Part Convergence Sublayer
CVS	Concurrent Versions System
DSS2	Digital Subscriber Signalling System No.2
ITU-T	International Telecommunication Union - Telecommunications Standardization Sector
ISDN	Integrated Services Digital Network
MTP-3	Message Transfer Part 3
OVOPS	Object Virtual Operations System
PDU	Protocol Data Unit
SAR	Segmentation And Reassembly
SSCF	Service Specific Coordinate Function
SSCOP	Service Specific Coordination Oriented Protocol
TOVE	Transparent Object-oriented Virtual Exchange
UNI	User-Network Interface
VE	Virtual Exchange

2. INTRODUCTION

This document describes software modules that belong to the TOVE project's 1996 Deliverables package. The package and this document will be distributed to project's participants in March 1997.

The document consists of several chapters, each chapter describing a single functional software entity that may include several modules (i.e., a protocol and its interface module). The chapters are meant to help reading the code and understand the software architecture but to use the code in other projects, one needs to go through the comments in the code and to have the corresponding specifications available.

Each chapter describes module's architecture, implementation-related issues and future development plans. A chapter also includes statistics about spent working hours and some basic code metrics. Not all the working hours are included in statistics of the individual modules. Table 1 shows approximations of hours not allocated to any module.

Meetings	Information gathering	Papers and presentations	Linux ATM and demo	Documentation	Total
160	160	320	320	320	1280

Table 1 Working hours not allocated to modules

TOVE project will provide support in case project's participants are going to use the software in their projects.

3. OVOPS++

OVOPS++ (Object Virtual Operations System ++) is a protocol framework for network programming that is based on Conduits+ and OVOPS. Conduit+ was developed by a team from Ascom Tech AG, University of Illinois and GLUE Software Engineering. OVOPS was developed by Telecom Finland and Lappeenranta University of Technology. OVOPS++ builds the Conduit+ framework on top of services provided by OVOPS. These services include, e.g., scheduling and timers. Currently, only a limited subset of OVOPS functions is used.

3.1 1 Introduction

This documentation describes some architecture and implementation solutions of the OVOPS++ framework that was implemented by the TOVE project. The implementation is based on an original PD software developed by Open Environment Software Oy. This documentation does not describe a general architecture of conduits because of well-documented papers about Conduits that are written by University of Illinois (see appendix) and Ascom Tech AG (see references).

3.2 2 Architecture

A layered protocol stack is made up of conduits, and messages are transferred through the stack. The framework has four kinds of conduits: protocol, mux, factory and adapter. Each conduit, except adapter, has a sideA and sideB to which other conduits can be connected. The protocol is a conduit used to implement one communication layer of protocol stack; it includes a pointer to a state machine and it holds protocol specific variables and timers. The mux is a conduit that connects one sideA conduit to any number of sideB conduits. Thus it is used to multiplex messages to one or more conduits. The factory conduit is used with the mux to create dynamically new conduits to the mux sideB in run time. The factory conduit has a prototype conduit (or conduit string) to be cloned and installed to the mux sideB. The adapter is a conduit which has only sideA. Using the adapter, the software implemented with conduits can communicate with the outer world.

Following figures (1-3) show relations between objects in the OVOPS++ protocol framework.

There are also a few examples of inherited classes and method names in figures. The first figure shows a conduit hierarchy, the second one shows a message and timer hierarchy. The third figure has rest of classes like classes for scheduling and trace. Grayish classes are OVOPS classes, and normal white ones indicate OVOPS++ framework classes.

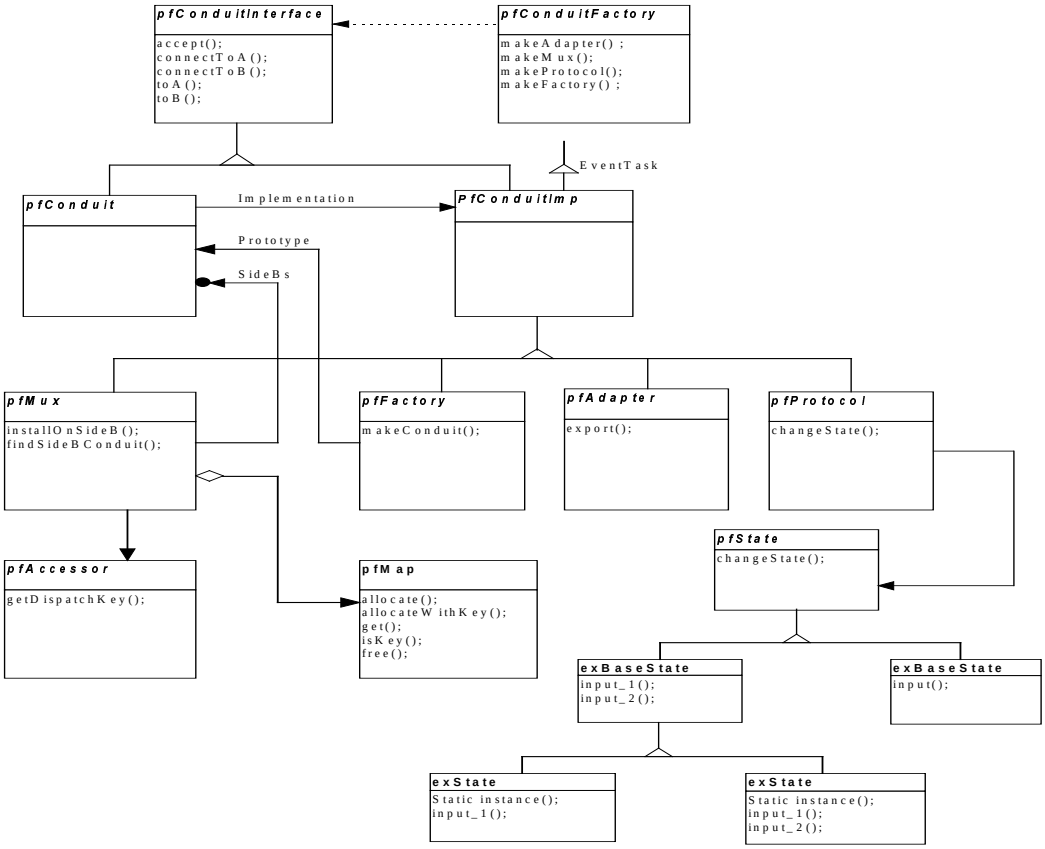

Figure 1. Relations between conduit objects

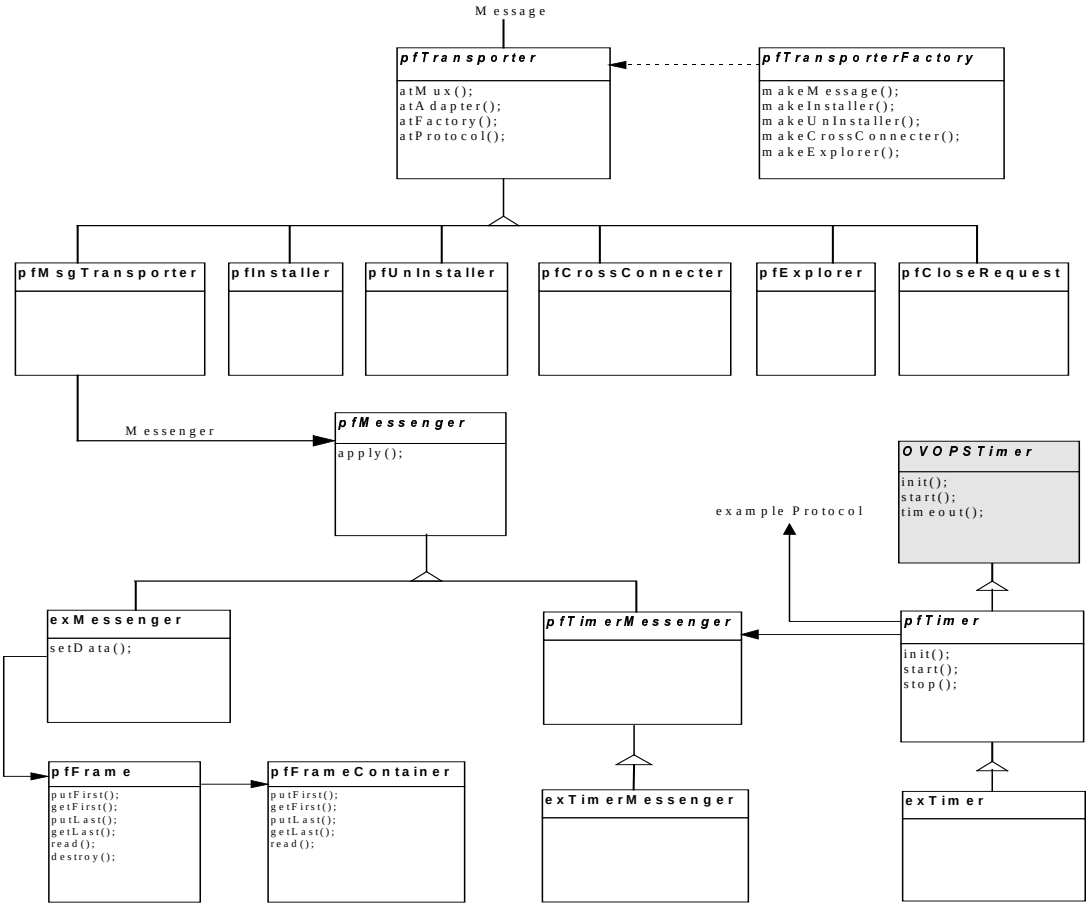

Figure 2 Message and timer hierarchy

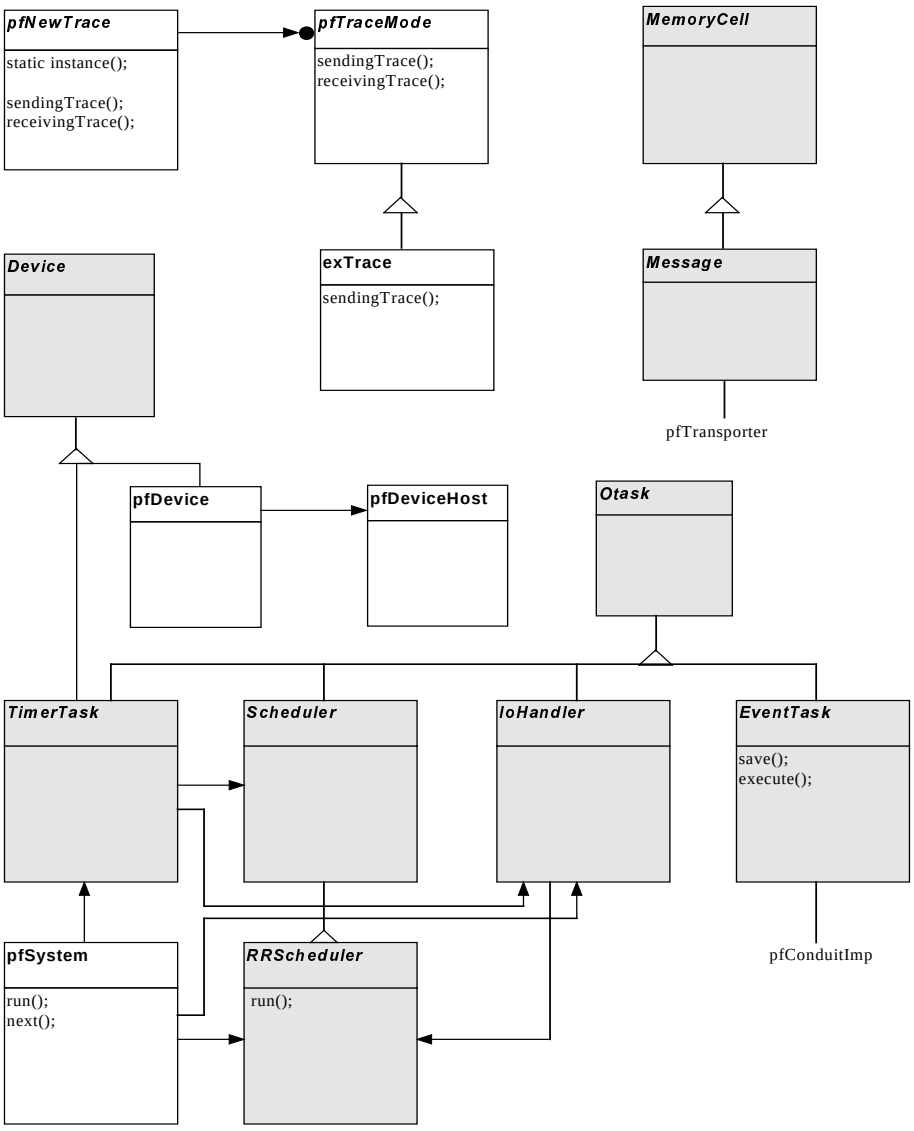

Figure 3 Relations between OVOPS and the framework

As these figures show, there is always at least one framework class (pf, protocol framework, as prefix) between OVOPS and user defined classes (ex, example, as prefix). In this way OVOPS was hidden from the user, and interfaces became simpler and clearer. This also makes the framework more flexible, because OVOPS classes behind the framework can be replaced with new implementations later like to a frame has already been done.

One major architectural detail, which is not mentioned in the Conduit+ paper, is the use of proxies. A conduit implementation is protected from a user inside the proxy. The user gets only the proxy to the corresponding implementation and use methods of the implementation through the proxy. The implementation includes a reference counter mechanism like a figure 4 shows.

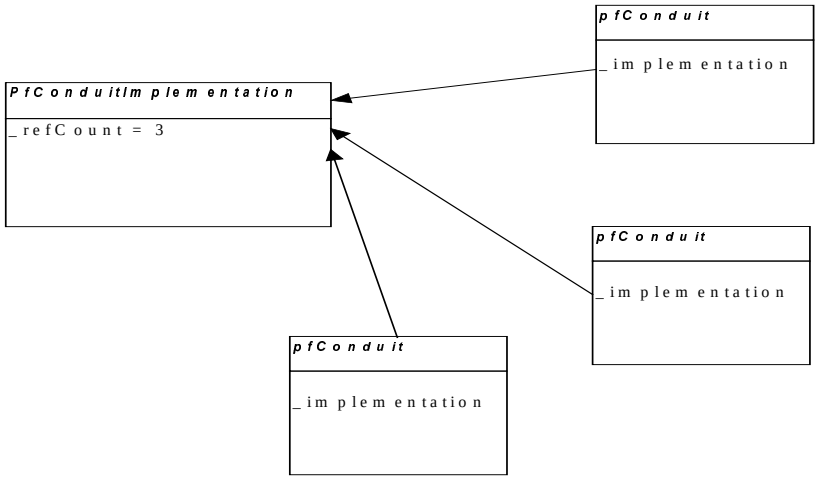

Figure 4 Reference counter and proxies

The implementation counts number of references (proxies) to itself. Each conduit implementation (except adapter) has conduits (proxies) on a sideA and sideB. When the last proxy to the implementation has been deleted also implementation itself will be destroyed. This feature is used, e.g., when protocol stacks (different connections) are uninstalled.

The following figure shows an example how OVOPS++ building blocks (conduits) can be connected to each other and thus implement an ATM signalling software.

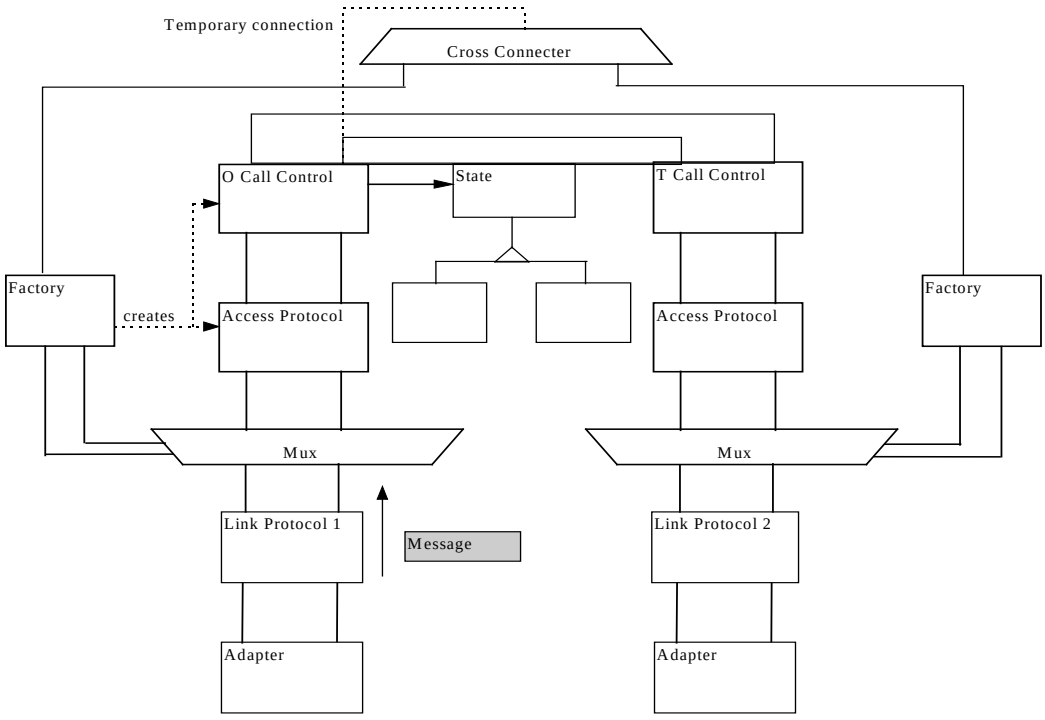

Figure 5 Example how the software of ATM switch can be build using OVOPS++

3.3 3 Implementation details

Following chapters describe implementation details and solutions that have not been explained or are solved in different way in the Conduit+ paper (appendix A).

3.3.1 3.1 Timers and scheduling from OVOPS

Timers and a scheduling mechanism are derived from OVOPS classes.

A protocol has timers as attributes or pointers to them. Protocols handle timers with start, stop and setTimeout methods in a pfTimer class. When the timeout occurs, the timer (inherited from the pfTimer) sends timeoutMessenger to the protocol that directs it to a right input in a state machine. Thus, timeouts were handled like any other messengers in the state machine.

The scheduling mechanism is mainly composed from two parts, tasks and messages. Conduits (implementations) are derived from an OVOPS EventTask, and messages (pfTransporter) are derived from an OVOPS Message, thus messages can be saved to the OVOPS scheduling queue. More detailed description of the scheduling can be founded in the OVOPS documentation.

3.3.2 3.2 Frame

All information chunks that the protocol processes and transfer are stored into frames within OVOPS++. Because this function is so essential in the information processing, the implementation must be efficient and secure. The basis of the efficiency is to avoid copying whenever possible (e.g., when a frame is transferred from a protocol instance to another). However, it must be secure at the same time, so that two protocols cannot write to same frame with unpredictable results, and on the other hand, when a frame is not referenced any more, the memory area is safely freed.

The pfFrame implementation is a reference-counting copy-on-write solution. The actual data is stored in pfContainer class instances that can be referenced by one or more pfFrame class instances. The user sees only instances of pfFrames, which hide the reference counting and copy-on-write details. If a copy is made of a frame, a new instance is made only of the pfFrame class and it is set to point to the existing pfContainer class. Now the container has two referencing frames, and if either of them writes to the frame (or otherwise changes it), a copy is made. When reference count of a container drops to zero, the memory area is freed automatically.

3.3.3 3.3 New transporters

A factory conduit has a prototype to be cloned when new conduits are installed on the mux sideB. If the prototype has more than one conduit, the first conduit is installed on the mux sideB and the last conduit of the conduit string has to be installed to an upper destination. An explorer transporter was implemented to travel through the prototype conduit string and find the last conduit from that string. This way the factory can connect

the last conduit sideB of the prototype string to the upper destination (to an upper mux for instance).

A closeRequest transporter is used to start an uninstalling process for the conduit string on the mux sideB. This transporter is send as asynchronous from a protocol to the mux. After receiving this closeRequest transporter the mux will send an uninstaller to the protocol to disassemble the conduit string and remove it from the mux sideB.

3.3.4 3.4 Multiplexer

Figure 6 shows how to use a multiplexer and how it works. The mux (multiplexer) have an accessor which gets a dispatch key from incoming message, and a map which keeps a collection of the mux sideB conduits. A PfMap uses an STL Map structure to store dispatch key and conduit pairs. In this OVOPS++ release, the pfMap object generates the new dispatch key when necessary and it supports only (long) integer values. In the Conduit+ paper has mentioned that accessors may include the mechanism to generate dispatch keys, thus user can define different criteria for generating keys.

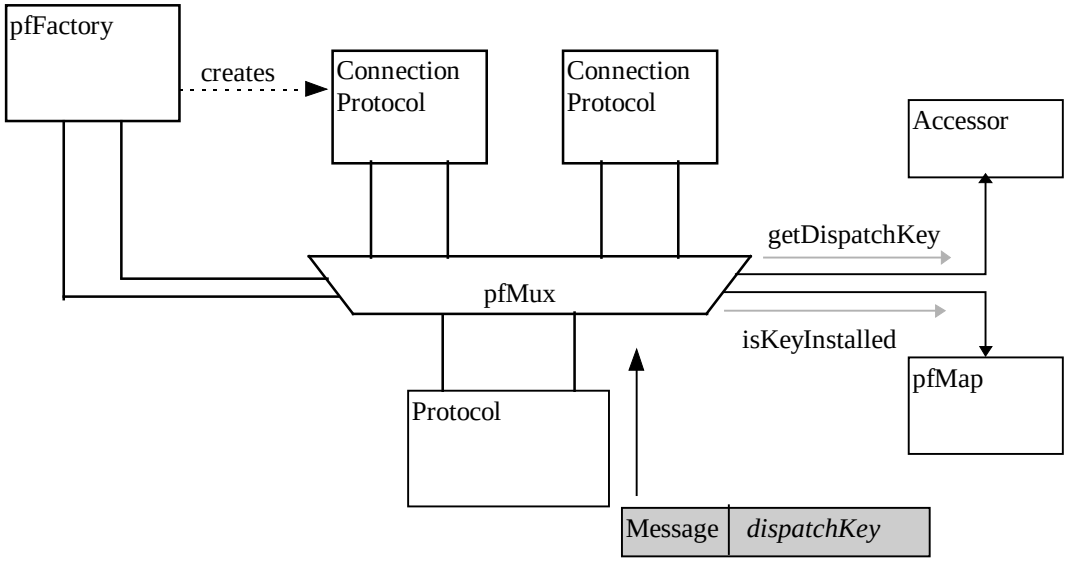

Figure 6 Function of multiplexer

If the dispatch key of an incoming messenger does not match any of conduits installed on the mux sideB, the incoming transporter is directed to the factory but only if message's installPermission flag indicates access to do it (installPrmission flag have to set e.g. when SETUP message is sent to the mux).

3.3.5 3.5 Other features in the framework

A device class is also provided (class pfDevice). It offers a general socket interface for network or UNIX interprocess communication. Its function is to tie a socket descriptor to OVOPS scheduling so that asynchronous events from the sockets (i.e., an incoming

packet) could be detected. For this reason pfDevice is built on OVOPS Device class. Note that pfDevice is not usable as it is, but it must always be inherited to implement a device for a specific class of file descriptors. This is because especially opening a file descriptor is always specific to the type of the descriptor (for example opening a disk file takes quite different parameters than opening a TCP/IP socket), so there cannot be any general open method. An other thing that the user of a pfDevice must take in accout is providing a callback interface for the device, if asynchronous messages to the device are possible. This is done by implementing pfDeviceHost interface to the class using the device (typically a class derived from pfAdapter or pfProtocol) and giving an instance of this class as a "host" class. The callback functions are called by the device when an asynchronous message arrives at the socket.

There are also classes for a trace system. All traces are called from the pfNewTrace class that implements a singleton pattern and delegates traces to an actual trace (mode) class stored in STL Map a link identifier as a key. The framework sends automatically trace calls when any message is sent or received from/to any conduit. The trace has a name information about message, sending/receiving conduit, protocol state and accessor. A user inherits an abstract pfTraceMode class to implement different message sequence charts , e.g., a chart with ASCII characters or some graphical diagrams.

3.4 4 Known bugs and flaws

There are two known "bugs" in the OVOPS++. The first one is in timer initializing. The software breaks if the user calls setMessenger or setTimeout methods before a setHost method, thus the host must be defined always before these methods can be called.

Another bug occurs when a transporter is stopped at the mux. This kind of situation may happen when the transporter has no install permissions set and the dispatchKey of an incoming messenger doesn not match any of mux sideB conduits, then the transporter is left without deleting. Normally the protocol on the mux sideB will destroy the transporter.

Transporters whose come to the mux from the sideA (from down) have to be send to the mux as synchronous, otherwise it is caused problems because a scheduling mechanism may give turn to the next message even if an installing process for a new connection (to mux sideB) is still unfinished.

3.5 5 Future development

There has been arisen a need for a state machine also in other conduits than the protocol. This means that the state pointer may be moved from the pfProtocol to the pfConduit class in the future. After this change, a user can implement some protocol specifications inheriting the mux conduit. Then same conduit has the accessor and state machine. The state machine also helps to implement better adapters.

In this moment the framework uses very much normal casting, but this is not suitable for complicated class hierarchies whose have several levels and use multiple inheriting.

Future study is considering to find a solution from double dispatching technique, but also C++ casts (e.g., dynamic_cast) will be taken into account.

As discussed in chapters 3.4 and 4, the multiplexer is not very robust. In generally the question is: which part of the framework is wanted to be constant and which part is intended to be inherited. In the future the dispatchKey may be generated in the accessor (like mentioned in the Conduit+ paper) and type of the key must be more general than integer, string for example. An error handling may be implemented in the state machine of the factory and an install permission flag can be removed in whole. These issues have to be researched carefully during the next development iteration of the OVOPS++.

Also the new timer and scheduling mechanism may be developed in the future. This means that the OVOPS code behind the framework will become unnecessary. Replacing the OVOPS code is going to be quite painless, due to own "adapter" classes between user and OVOPS classes in the framework.

There is a possibility to enhance the performance of the frames. At the moment all the data areas are allocated and freed every time a container is created or destroyed, but a more efficient approach would be adding the free memory areas into private free list. Also methods to split and combine frames could prove useful.

3.6 6 User's guide

This chapter gives short introduction with code examples how to use the OVOPS++ framework. Following simple code examples correspond to the protocol stack showed in the figure 6.

3.6.1 6.1 Implementing of protocols and states

First code examples define a header and implementation files to a protocol connected to the mux sideA in the figure 6. It is very simple protocol, which have one timer.

```
//Description:
//    Define header file to a protocol connected to the mux
//    sideA (downward). The protocol has one timer.
//

#include "pf/protocol.h"

class Protocol : public pfProtocol
{
    public:
        Protocol(void);
        virtual ~Protocol(void);
        void setT1timeout(pfUlong msec_);

        void sendConnectMessageToB(void);
        void sendReleaseMessageToB(void);

    private:
```

```
        pfTimer _timerT1;
};
```

In the constructor of the implementation code example initializing of the timer can be seen. First host is set for the timer, then the timeout messenger and last timeout value. The protocol is also initialized to the ReadyState.

The protocol has also methods to send different messages to the mux (on sideB of the protocol).

```
//Description:
//    Define implementation file to a protocol connected to the //
//    mux sideA (downward). The protocol has one timer.
//

#include "exampleprotocol.h"
#include "examplemessages.h"

Protocol :: Protocol(void)
    : pfProtocol()
{
    _timerT1.setHost(this);

    pfTimerMessenger *messenger = new TimeoutMessage;
    assert(messenger != 0);
    _timerT1.setMessenger(messenger);
    setT1timeout(100);

    changeState(protocol_, ReadyState::instance());
    return;
}

Protocol :: ~Protocol(void)
{
    return;
}

void Protocol :: setT1timeout(pfUlong msec_)
{
    pfUlong usec = (msec_ % 1000) * 1000;
    pfUlong sec = (msec_ - (usec / 1000)) / 1000;
    _timerT1.setTimeout(sec, usec);
    return;
}

void Protocol :: sendConnectMessageToB(void)
{
    ConnectMessage *messenger = new ConnectMessage;
    assert(messenger != 0);
    toB(messenger);
    return;
}

void Protocol :: sendReleaseMessageToB(void)
{
    ReleaseMessage *messenger = new ReleaseMessage;
    assert(messenger != 0);
    toB(messenger);
```

```
    return;
}
```

Next protocol is a connection protocol. It has a clone method and a copyconstructor because it is cloned by the factory when new connection is created and installed to the mux sideB. Below is the header file of that protocol example.

```
//Description:
//    Define header file to a connection protocol connected
//    to the mux sideB (upward). The protocol have to be a
//    clone method because of factory will clone it and install
//    these instances to the mux sideB.
//

#include "pf/protocol.h"

class ConnectionProtocol : public pfProtocol
{
    public:
        ConnectionProtocol(void);
        ConnectionProtocol(const ConnectionProtocol &other_);
        virtual ~ConnectionProtocol(void);
        virtual ConnectionProtocol *clone(void) const;

    private:
        void sendAcknowledgeMessageToA(void);
};
```

Following codes define a state machine for the connection protocol. The first file describes a header file of a base state with all possible inputs to the connection protocol state machine. If the connection protocol had any timers, also inputs for timeout messengers would be in this file.

```
//Description:
//    Define header file to a base state for a connection
//    protocol state machine.
//

#include "pf/state.h"

class pfMessenger;

class ConnectionState : public pfState
{
    public:
        ConnectionState(void);
        virtual ~ConnectionState(void);

        virtual void ConnectMsgAct(ConnectMsg *messenger_,
                                   pfProtocol *protocol_) const;
        virtual void ReleaseMsgAct(ConnectMsg *messenger_,
                                   pfProtocol *protocol_) const;
};
```

Next two files (header and implementation) show an example of actual states (ConnectionIdle) in the state machine joined to the connection protocol. The state was

implemented with singleton pattern, and it has one rewritten input method (ConnectMsgAct).

```
//Description:
//    Define header file to a idle state for a connection
//    protocol state machine.
//

#include "pf/state.h"

class pfMessenger;

class ConnectionIdle : public ConnectionState
{
    public:
        static ConnectionIdle *instance(void);

        virtual void ConnectMsgAct(ConnectMsg *messenger_,
                                   pfProtocol *protocol_) const;

    protected:
        ConnectionIdle(void);
        virtual ~ConnectionIdle(void);

    private:
        static ConnectionIdle *_only;
};
```

When the ConnectionIdle state is received the ConnectMsg message, it sends an AcknowledgeMsg message to the mux and change the state to the ConnectionActive state.

```
//Description:
//    Define implementation file to a idle state for a connection
//    protocol state machine.
//

#include "pf/messenge.h"
#include "idlestate.h"
#include "exampleprotocol.h"

ConnectionIdle *ConnectionIdle :: _only = 0;

ConnectionIdle *ConnectionpIdle :: instance(void)
{
    if (_only == 0)
    {
        _only = new ConnectionIdle;
        assert(_only != 0);
    }
    return _only;
}

ConnectionIdle :: ConnectionIdle(void)
    : ConnectionState()
{
    return;
}
```

```
ConnectionIdle :: ~ConnectionIdle(void)
{
    _only = 0;
    return;
}

void ConnectionIdle :: ConnectMsgAct(
    ConnectMsg *messenger_,
    pfProtocol *protocol_) const
{
    ConnectionProtocol *p = (ConnectionProtocol*) protocol_;
    p->sendAcknowledgeMessage();
    changeState(protocol_, ConnectionActive::instance());
    return;
}
```

3.6.2 6.2 Implementing of messages

The following code of implementation file shows how to implement simple messages. User have to rewrite an apply() method from which the right state input is called. These example messages doesn't include any data thus they have only apply() methods and none attributes.

```
//Description:
//    Define implementation file to messages transferred
//    between the example Protocol and ConnectionProtocol.
//

#include "examplemessages.h"

void ConnectMessage :: apply(pfState *state_,
                             pfProtocol *protocol_)
{
    assert(state_ != 0);
    ConnectionState *s = (ConnectionState *) state_;
    s->ConnectMessageAct(this, protocol_);
    return;
}

void ReleaseMessage :: apply(pfState *state_,
                             pfProtocol *protocol_)
{
    assert(state_ != 0);
    ConnectionState *s = (ConnectionState *) state_;
    s->ReleaseMessageAct(this, protocol_);
    return;
}
```

3.6.3 6.3 Implementing of accessors

A pfAccessor class in the OVOPS++ framework has two pure virtual methods, whose user have to implement. The clone() method of the accessor is needed when the mux is cloned. The purpose of the getDispatchKey is to get a protocol specific identifier from the message. Due this key the mux can store conduits on its sideB.

```
//Description:
//    Define implementation file to accessor
//

#include "exampleaccessor.h"
#include "examplemessages.h"

Accessor :: Accessor(void)
    : pfAccessor()
{
    return;
}

Accessor :: ~Accessor(void)
{
    return;
}

Accessor *Accessor :: clone(void) const
{
    Accessor *newAccessor = new Accessor(*this);
    assert(newAccessor != 0);
    return newAccessor;
}

pfKey Accessor :: getDispatchKey(
     const pfMessenger *messenger_) const
{
    pfKey key = ((Messages *)messenger_)->getConnectionKey();
    return key;
}
```

3.6.4 6.4 Example of main function

The last code is an example of main function to be used to create the protocol stack by connecting conduit to each other. The mux and factory are created with pfConduitFactory thus the accessor is set as a parameter when the mux is created and a proxy to the connection protocol when the factory is created. The Factory use this proxy as prototype for new connections.

```
//Description:
//    Define main function to create protocol stack and
//    send one message.
//

#include "pf/factory.h"
#include "pf/cfactory.h"
#include "pf/mux.h"
#include "pf/system.h"

#include "exampleprotocol.h"
#include "exampleaccesor.h"
#include "connectionprotocol.h"
#include "examplemessages.h"

int main(void)
{
```

```
    pfId id = 1;

    // Create conduits
    Protocol *exampleProto = new Protocol();
    assert(exampleProto != 0);
    pfConduit exampleProxy(exampleProto, 1);
    exampleProxy.setId(id);

    ConneectionProtocol *conProto = new ConnectionProtocol();
    assert(conProto != 0);
    pfConduit conProxy(conProto, 1);
    conProxy.setId(id);

    pfConduitFactory *absFac = pfConduitFactory::instance();
    pfConduit factoryProxy = absFac->makeFactory(conProxy, id);

    Accessor *accessor = new Accessor;
    assert(accessor != 0);
    pfConduit muxProxy = absFac->makeMux(accessor, id_);
    muxProxy.setId(id);

    // Connect conduits
    exampleProxy.connectToB(muxProxy);
    muxProxy.connectToA(exampleProxy);
    muxProxy.connectToB(factoryProxy);
    factoryProxy.connectToA(muxProxy);

    // Create trace
    pfNewTrace::instance()->setTrace(exampleTrace::create(), id);

    // Register conduits to the trace
    exampleProxy.setTraceOn();
    conProxy.setTraceOn();
    factoryProxy.setTraceOn();
    muxProxy.setTraceOn();

    // Send message
    exampleProxy.sendConnectMessageToB();

    // Run the system !!!
    pfSystem::instance()->run();

    return 0;
}
```

In the main function there is also initializing for a trace. The link identifier is set to each conduit and then the example trace object (inherited from pfTraceMode) is set to the common trace object (singleton) an identifier as a key. Conduits, to be traced, are registered to the trace object.

Finally the example protocol sends a message to the mux, and the scheduling system is started.

3.7 7 Statistics

The following table shows estimated figures of hours used to develop the OVOPS++ protocol framework during the first year of the TOVE project. **320 hours were used outside the project to implement the first prototype of OVOPS++.** These hours are included in the table below.

Activity	Research	Design	Coding	Reviews	Total
Duration (h)	200	130	190	60	580

Table 1 Duration of activities

Next table shows a size of the framework as code. Numbers describes metrics of the framework including the original PD software code but no the OVOPS code by Lappeenranta University of Technology.

Lines Of Code (LOC)	Number of files	Number of classes
4109	37	30

Table 2 Metrics

3.8 8 References

H. Hueni, R. Johnson, R. Engel. A Framework for Network Protocol Software, OOPSLA'95 Proceedings, Austin,1995

R. Engel. Signalling in ATM networks: Experiences with an object-oriented solution. In International Phoenix Conference on Computers and Communications, IEEE, 1995.

E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Addison-Wesley, 1995.