Author: Vesa-Matti Puro

Document: OVOPS++

Date: 24/03/1998

Version: 0.1

This document contains ideas about conduits and its use for implementing TOVE software. Some ideas describe the present situation, some ideas are included here to be used in the future development of the OVOPS++ framework. Also some aspects of TOVE software are discussed.

OVOPS++ Protocol Framework

INTRODUCTION

The OVOPS++ Protocol Framework is designed to help implementation of layered protocol software. The framework provides classes that can be used or extended when implementing protocol software. In co-operation with OVOPS++ Scheduling Framework the framework also provides ready to use flow of control to be used to control different aspects of protocol software such as events, asynchronous tasks, timers and devices. The framework provides classes to structure protocol software and to model information flowing in the software. The classes providing structure are called conduits and the classes modelling information are called visitors and messengers.

In addition to the original framework, new problem domain terms are introduced: pfPDU (pfFrame), pfInformationElement (pfStorage) and pfPrimitive (pfMessenger).

CONDUITS

There are four conduit categories in the framework. The conduit classes are pfAdapter, pfFactory, pfMux and pfProtocol. Each conduit have one or two sides (A or A and B).

Conduits have a built-in automatic memory management. A Conduit is destroyed when it is not used any more or when that is requested.

The conduits are accesses through proxies. When a conduit is created, a pfControl proxy is given to user. This proxy can be used to create a pfProxy that can be used to access the side A or the side B of the conduit.

Problem: how to know where a transporter is coming and how to connect proxies. Solution can be that there are proxies to different sides of the conduit. This requires acceptFromA and acceptFromB as well as connectA and connectB methods in the conduit. Separate accept methods remove the need of comparisons where the visitor is going. The proxy can be specialized to side A proxy by using method pointers as a strategy.

In the case of the pfMux, the createB method can create proxies to different indexes of side B of the multiplexer. This difference is

hidden to the implementation. Is there a problem than if we create a proxy to side B, do we need the index at that time, or later?

In the case of pfAdapter, the semantics should be defined. Do we create a null conduit in that case or do we throw an exception.

The disconnect method in the proxy or the destruction of the proxy is the opposite operation for creation of the proxy.

```
class pfControl
{
    public:
        pfControl (void);
        pfControl (const pfControl &other_);
        virtual ~pfControl (void);

        pfProxy createA (void) throw (pfCannotCreate);
        pfProxy createB (void) throw (pfCannotCreate);
};
```

connect: connect the other object to us, for example if we are a proxy to A side, and the other is a proxy to B side, then connect the other B side to our A side. Then you have to connect it other way, because connection is oneway. You may also check that you don't connect your own A side to your own B side.

Question: can more than one connect be done with one proxy. Or do we need to throw pfAlreadyConnected exception in that case.

destroy: deletes all conduits until the B side of a pfMux or an pfAdapter is reached, i.e. deletes the independent sub-stack. It has no sense to delete just ourselves.

disconnect: releases the implementation inside the proxy if the reference count goes to 0.

Adjacent protocol layers form an independent protocol sub-stack. An independent (and identifiable) protocol sub-stack is the unit that is destroyed as a unit. An independent protocol sub-stack can be created by factory. An identifiable sub-stack is identified in the trace output as an independent entity.

Problem: the connection is oneway, is there a pointer to the guy who points to us? This would be needed if we want to go away and inform our proxies that we are gone.

```
class pfProxy
{
    public:
        pfProxy (void);
        pfProxy (const pfProxy &other_);
        virtual ~pfProxy (void);

        void connect (contst pfProxy &conduit_);
        void destroy (void);
        void disconnect (void);

        void accept (pfVisitor *transporter_) throw (pfCannotAccept);
```

```
};
```

PFADAPTER

Problem: when writing an API, you need a specific method interface, conduits have an automatic memory management, API cannot have it because that have other rules for it.

The PfAdapter can be used as an API or as a device driver. Device drivers are supposed to be inherited from pfAdapter. APIs are supposed to created by giving a callback object to the adapter. PfAdapter does not delete the callback object when it is deleted.

There could be open, close and error request and indication messengers for controlling the adapter. Then the function of the adapter would be dynamic.

```
class pfAdapter
{
    public:
        static pfControl createAdapter (pfAdapterCallback *callback_)
            throw (pfCannotCreate);
        virtual ~pfAdapter (void);

    protected:
        pfAdapter (void);
        pfAdapter (const pfAdapter &other_);
};
```

pfAdapterCallback processes the given messenger synchronously. The contents of a messenger is copied if it is needed. PfAdapterCallback gives messengers to pfAdapter to be transmitted further. The pfAdapter can be disconnected from the callback object by calling the disconnect method.

```
class pfAdapterCallback
{
    public:
        virtual ~pfAdapterCallback (void);
        virtual void callback (pfMessenger *messenger_);
        virtual void connect (const pfConduit &conduit_);
        virtual void disconnect (void);
};
```

```
class ExampleAPI : public pfAdapterCallback
{
    public:
        // from framework
        virtual void callback (pfMessenger *messenger_);

        // from API
        void MethodA (void);
        void MethodB (void);
        void MethodC (void);
};
```

PFFACTORY

How the prototype is given to pfFactory? The factory can be given a list of conduits that is cloned when a new dynamic instance is needed. The list is first traversed and then the conduits are connected. Maybe the contents of the list can be restricted. Only pfProtocols are allowed with predefined system for connecting the protocols. The list is stored in factory and the clone method is implemented in factory.

The new instance of sub-stack is given a specific identifier when it is cloned from the prototype. That identifier is the identifier of the mux added by the index given for the new connection.

```
class pfFactory
{
    public:
        typedef list<pfControl> ConduitList;
        static pfControl createFactory (const ConduitList
&conduitlist_)
            throw (pfCannotCreate);
};
```

PFMUX

How the creation of indexes and the extraction of indexes are abstracted. Is accessor used? Maybe we can use default accessor? Then we need parameters for the default accessor, or we need the accessor. Or maybe we can have a default accessor that can be given to the mux, so we need only one create method.

Needed limits: start of the indexes, increment, end of the indexes, the number of the indexes that can be used at one time.

Problem: the indexes are allocated half and half local and remote. How this algorithm works? Do we need to parametrize algorithm or just the parameters.

The pfMux need interface how the adjacent protocol can access some information about the mux.

When we do a crossconnection, we need a new index if the messenger comes to the factory from the factory side B and we want to use the old index if the messenger comes to the factory from the factory side A.

```
class pfMux
{
    public:
        static pfControl createMux (const string &name)
            throw (pfCannotCreate);
};
```

PFPROTOCOL

How the protocol sees the contents of the map of the mux. There could be an interface for that or a common map for them.

```
class pfProtocol
{
    public:
        static pfControl createProtocol (void)
            throw (pfCannotCreate);
};
```

VISITORS AND MESSENGERS

The direction of the messenger is noticed and recorded at a proxy that is specific to certain side of the conduit. In most cases the proxy gives the visitor to right method using a method pointer as strategy. If the direction is needed to recorded in the visitor that can be done in special cases.

The direction is needed in pfAdapter, pfFactory and pfMux. The direction may be needed in pfProtocol if the messengers need to be checked (up primitives come from the right side, etc.).

```
class pfVisitor
{
    public:
        void atAdapter (pfAdapter *adapter_);
        void atFactory (pfFactory *factory_);
        void atMux (pfMux *mux_);
        void atProtocol (pfProtocol *protocol_);

        void incrementReferenceCount (void);
        long decrementReferenceCount (void);
};
```

```
class pfMessenger
{
    public:
        void apply (pfProtocol *protocol_, pfState *state_);

        void incrementReferenceCount (void);
        long decrementReferenceCount (void);
};
```

Visitor is a proxy for the messenger in the sense of who controls the memory management. We want to implement reference counting to messengers as well it is already implemented in the conduits.
The message queue in the protocol is also a proxy for messengers.

Conduits are proxies for the visitors and visitors are reference counted as well. Synchronous visitors are given a reference count that prohibits them from being released by conduits. Synchronous visitors are created and deleted usually by using automatic local variables.

Reference counting works as follows: when the reference counted object is created the reference count is set to 0, when the pointer is used the reference count in incremented (accept), when the pointer is not needed any more (end of runcallback) the reference count is decremented and after that the object has to be deleted if the reference count reaches 0.

The idea: if somebody uses a pointer to reference counted object, it increments the reference count first and after it no longer needs it the decrement the reference count and deletes if it notices that this object is no longer needed.

Cloning increments the reference count (if no copy is created). Maybe the messages are not intended to be cloned but used with reference

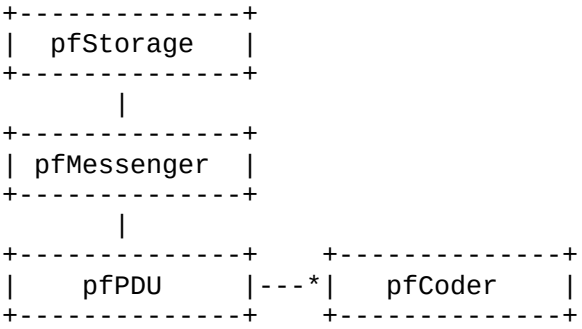
counting. This also means that if we use reference counting, we cannot modify the messenger any more.

Multicast or Broadcast visitor: It would be broadcast visitor, because the visitor that is copied to all available connections is needed more than multicast visitor that is copied to a list of destinations in the multiplexer.

PDU

The PDU would contain information elements in the storage. Then the operation for forming PDU from a primitive would be easy. The PDU would contain also a list of information element encoder/decoder objects for each information element for encoding and a map of the same objects for decoding (information element type is the key of map). These objects could be singletons.

How mandatory information elements and the right order for information elements are handled (in coding and in decoding).

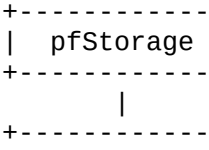


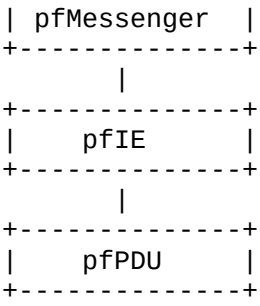
The pfCoder contains methods for operating between frame and storage. The basic methods are encode, decode, frameToStorage and storageToFrame, other methods can be defined for BCD numbers, octect arrays, or extension bits.

```
class pfCoder
{
    public:
        void encode (f, s);
        void decode (f, s);

        void frameToStorage (f, s, number_of_bits, name); // extract
        void storageToFrame (f, s, number_of_bits, name); // insert
};
```

The pfCoder classes can be implemented as a chain of responsibility pattern and as a member variables of pfPDU.

Fist we considered the following arrangement but that was too complex.





Each specific PDU base class would have a static method for creating PDUs from frame and metdhod fro creating a PDU from primitive.

```
m = static xPDU :: decode (f);

m = xPDU :: crate (p);
```

If a decoding error happens an invalid messenger could be returned (pfInvalid or xInvalid) or an exception could be raised.

PRINCIPLES

Could the memory allocation be done at once when the system is started.

All methods use exceptions to inform about run time errors.

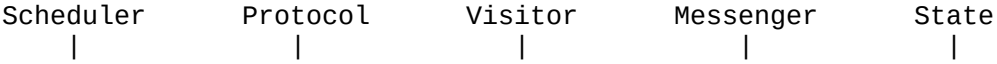
The methods that are not available at that class can throw pfNotImplemented method. This is maybe better than to use many different exceptions such as NotCloneable, NotMovable, etc.

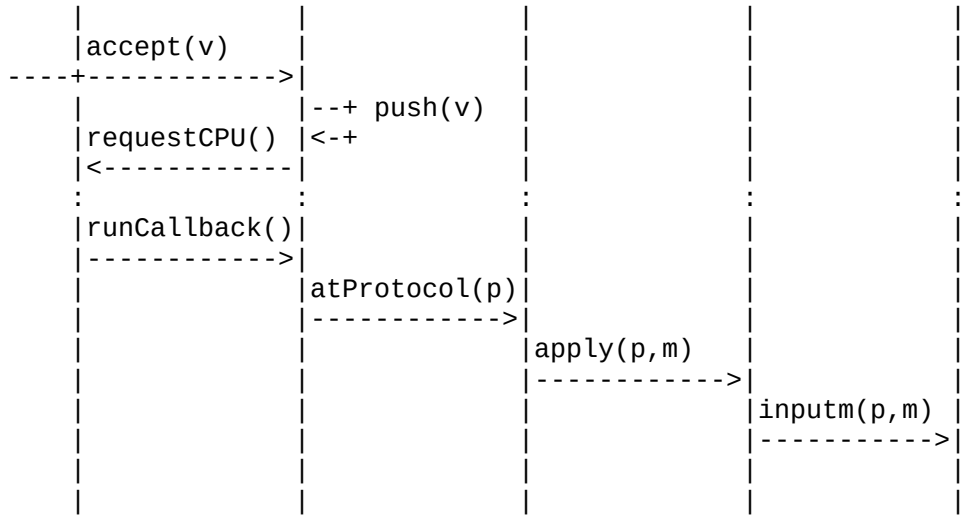
ERROR HANDLING

There are different levels of security. There are checks of validity of information when each level is accessed and inside each level checks are not performed. The level is selected based on the easiness of performing checks. A conduit can be that kind of level, proxy for other conduit, and implementation for the state machine. The check is performed again if that kind of error could happen inside the level. A conduit does not propagate exceptions out of the conduit at run-time.

Software errors are locked to singleton Error object that is always present. The following information is needed: severity, instance, location within the instance and the error code. Errors and reaching limitations should be separated.

Could we use an error counter inside a conduit and forward an error report only after certain number of errors. This could be suitable for dynamic device errors, but not for more severe type of errors.





TRACE

Trace can be used to watch static and dynamic instances.
The level of trace needed should be configurable. We need an user interface for this.
The trace should be as automatic as possible.
The trace can be grafigal or textual.
The trace can be directed to TOVE Java UI console.

Trace is based on identifying object by identifier that is automatic.
Selections, which:
- O protocols
- O messages
- O functions inside protocol etc.
- O calls are traced

Could we learn something from debugger, breakpoints, watchpoints, etc.

Identifier is a vector of integers. The value –1 is a special value for unknown. (3 7) could mean the call number 7 at atm link 3.

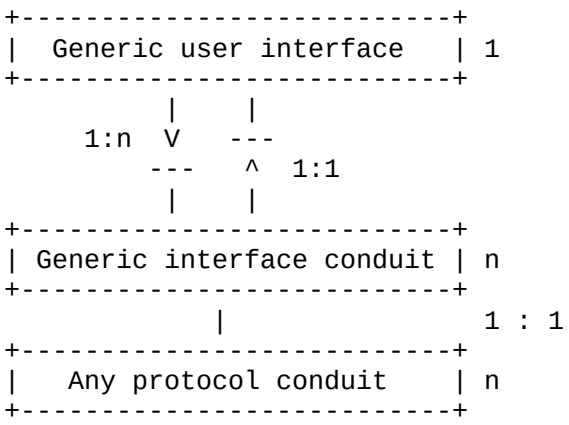
There should be means to inspect (and change) the contents of the state of the protocol, i.e. variables (storage), current state and timers. If the change of variables is allowed, how we specify the valid range of values.

LOG

This is standard thing to have in protocols and telecommunications equipment. There are standardised format for this. OMG notification service should be checked for information as well as ITU-T LOG standard.

USER INTERFACE

For testing purposes an user interface is needed. The user interface is separated from the protocol software using CORBA IDL interface.



Generic user interface is connected to n generic interface conduits (this conduit needs a name). Each generic interface conduit is connected to one (and (maybe) the same) generic user interface conduit.

The user can:
- O identify interfaces
- O select one interface
- O query the interface primitives (list)
- O select one primitive
- O query the parameters for the selected primitive (describe)
- O give values for the parameters
- O send the selected primitive (send)
- O observer received primitives (send)

OtMsg interface is used.
Generic user interface conduit needs a name (for user).
Maybe the system needs a name that could be acquired from the system conduit.

TESTING

The code should be finalised, commented and inspected before testing.

The protocols should be prepared for testing by:
- O set correct parameters
- O set up correct installation
- O create interface to control the protocol
- O create interface to connect the protocol to tester using generic interface conduit
- o create conduit to handle decoding and encoding the parameters

Find the abstract test suites needed: GSMP, UNI 3.1 and SSCOP.

RUNTIME ERRRORS

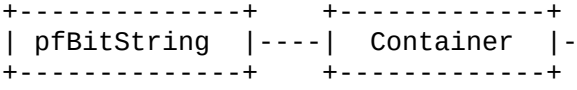
Sources could be for example from memory allocation or device errors.

```
xState :: inputM (p, m)
{
    xp = dynamic_cast<xProtocol>(p);
    xp->op1 ();
    xp->start(“T1”);
    try
    {
        v = xp->get(“A”);
        m->set(“A”, v);
    }
    catch
    {
    }
    xp->sendM(m);
    xp->toS2();
};
```

Suggestions:
1. there is one try catch structure and then default funtionality for each input method
2. use macro to hide the dynamic_cast (throw can be added, or changed to static cast)

pfBitString

There is a problem to represent the contents of encoded representation of PDUs in a layered protocol implementation. How we can add protocol control information in the head of the message in a dynamic way without copying the data unnecessarily.



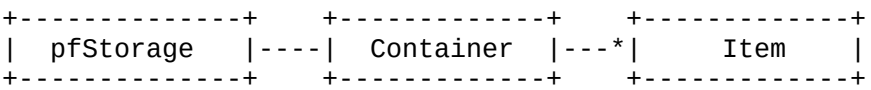

```
class pfBitString
{
    public:
        pfBitString create (void);
        void release (void);
        void putBits(pfULong count_, pfULong value_);
        pfULong getBits(pfULong count_);

    private:
    class Container
    {
        public:
        …
    };
};
```

The pfBitString acts as a proxy to the container which is shared with a several proxies. The container is reference counted. The proxy can give up the association to a container with a release method. The pfBitString included a write token that enables the proxy to write to the container. If a new proxy is created from an old one, the new one gets the write token, and the old one gives it away. A proxy can obtain the write token also if the reference count is one, otherwise the copying is needed.

What does the operation= in pfBitString for the token?

PfStorage

We need to define the semantics associated with the storages inside a storage. What happens if we want to get a storage.




```
class pfStorage
{
    public:
        PfStorage *getStorage (string name_);
};
```

Problem: The current fetch method of storage replaces, we would need an operation that keeps the old contents, replaces the old values when needed and brings new data items as needed. Jari had an idea of methods:
extendCopy (copies n new items),
copyVariable (copies 1 new item),
copyIfPresent (handles 1 optional item) and
getIfAvailable (copies if the member is defined and it has a value).

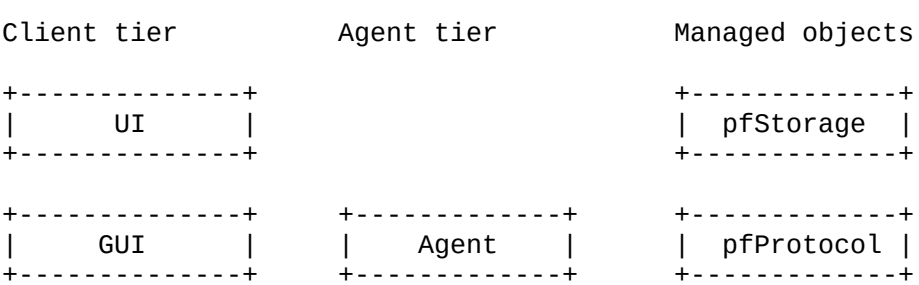
pfTimer

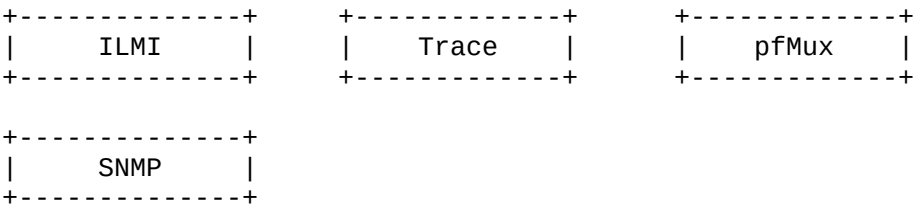
There is a different way of implementing a timer.

```
class pfTimer
{
    public:
        pfULong init (pfULong timeout_);
        void start (pfULong identifier_, pfULong ms_);
        void stop (pfULong identifier);
        void destroy (pfULong identifier);
};
```

TOVE RELATED STUFF

MANAGEMENT





Could trace be implemented as a SNMP trap. Then Agent and trace would be the same.

SNMP operations

1. get
2. get-next
3. set
4. trap

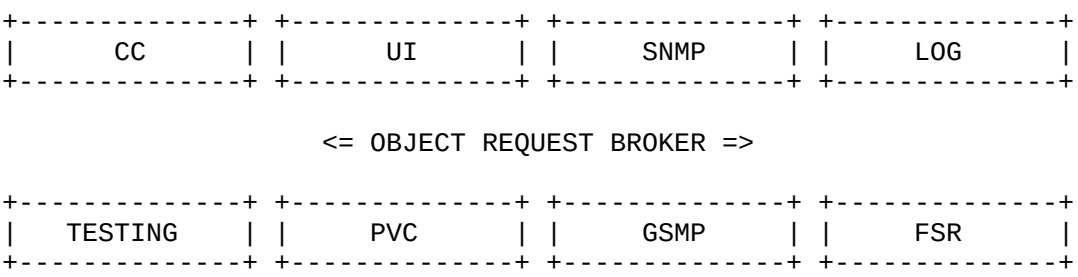
CMIS operations

1. get
2. replace
3. replace with default
4. add member
5. remove member
6. create
7. delete
8. action

Check www.adventnet.com for info.

What are the products we want to use?

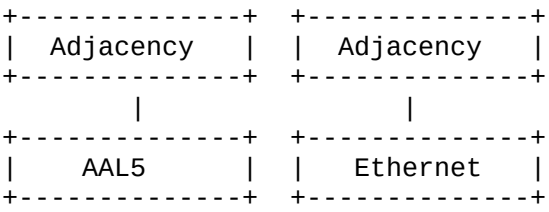
TOVE ARCHITECTURE



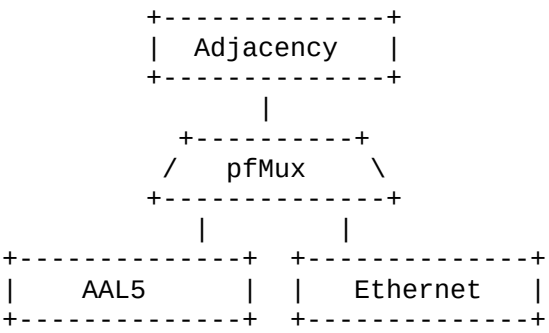
GSMP

Currently GSMP version 1.1 as in RFC 1987 is implemented. A new version 2.0 of GSMP is issued as RFC 2297. The main issues there are the addition of QoS and numerous corrections and modifications to the protocol. A support for ethernet is also added.

We could implement the support for ethernet encapsulation by supporting compile time selection of the protocol.



Or we could support configurable support for both AAL5 and ethernet encapsulation at the same time.



The switch should support the following use cases.

1. start of a switch
2. addition of a port
3. deletion of a port
4. port goes up
5. port goes down
6. addition of a connection
7. deletion of a connection
8. stop of a switch

Who about the the following ideas.

If port is brought up, the signalling cpcs is started.
If the cpcs has an error, that is reflected to uni.
If uni is informed of an error, that is reflected to calls.
If call is informed of an error, that is reflected to connections.
If port is taken down, the connections are informed.
If connections are taken down, the calls are informed.
Etc.

The following ideas about the attributes of a port and switch are derived from information in GSMP messages.

Each port has the following information.

1. Port number
2. VP switching
3. Multi-point support
4. QoS support
5. VPI range
6. VCI range
7. Transmit rate
8. Receiver rate

9. Port type
10. Port status
11. Line status
12. Physical port number
13. Physical slot number
14. Priorities

Each switch has the following information.

1. Version
2. Type
3. Name (producer and serial number)

THE TOVE SOFTWARE

In the implementation of TOVE software we want to use the following things.

1. Gather information about the current status and features of ATM switch software based on standardisation and commercial products
2. Develop a superset of requirements and select a subset for the software
3. Develop a superset of use cases and select a subset that must be supported

Requirements
1. supports SVCs
2. supports PVCs
3. supports VCCs
4. supports VPCs
5. supports UNI 3.1
6. supports Q.2931
7. supports BISUP
8. detects, reports and handles failures in call control

Use cases
1. start a switch
2. stop a switch
3. add a port
4. delete a port
5. connection from user to user