CVM Channel

Software Design Specification

Alexander Kogtenkov, kwaxer@aha.ru
Eugene Melekhov, eugene_melekhov@mail.ru
Dmitri Kondratiev, dkondr@bigfoot.com

Abstract

Packet flow is the set of the packets incoming to or outgoing from the network node selected in accordance to some condition. Channel abstraction defines the processing function performed on the packet flow. This document describes the channel architecture and channel API.

TABLE OF CONTENTS

Introduction

In general network node is a packet processor. As the packet processor the network node interested in packets, which it routes from place to place based on packet header information. The essential characteristic shared by all packet processors is the motion of packets. Packets arrive on one network interface, travel through the packet processor's *forwarding path*, and are emitted on another network interface. Channel describes the processing function to be performed on the given packet flow inside the channel. In this document we describe our channel architecture implementation and the API, which allows active applications to use channels.

- **Scope**

The Channel API is the part of the CVM.

- **References**

1. Architectural Framework for Active Networks Ken Calvert, lead author
2. Node OS and Interface Specification Larry Peterson, lead author
3. Packet Processing Framework Software Functional Specification.
4. The Click Modular Router. http://www.pdos.lcs.mit.edu/click/
5. Router Plugins: A Modular and Extensible Software Framework for Modern High Performance Integrated Services Routers. (1998) Dan Decasper, Zubin Dittia, Guru Parulkar, Bernhard Plattner… SIGCOMM. http://citeseer.nj.nec.com/decasper98router.html
6. ARM Transport - Active Reliable Multicast Transport Protocol.

Problem Definition

Active network node interface is designed around the idea of network packet flows: packet processing is done on a per-flow basis. Channels play a central role in supporting flow-oriented design. So, the channel design affects the whole CVM design and implementation and it should be flexible, modular and efficient. In our model Channel is the program that works on the packet flows inside the channel. Packets gets to the channel from the ingress or egress, packets are emitted from the channel to the network. So in general channel can have three associated packet flows – incoming packets from the ingress or egress and outgoing packets. Channel API allows creating channels, modifying channel, looking for the channels that meet some criteria. Channel is owned by some active application, so it's AA resource with clean ownership. Channel has associated Access Control List (ACL) which defines rights of other AA to use and/or modify the channel. At the moment ACL only defines shared, not shared channel status (in the next versions we can provide fine grain sharing).

Typical use of the Channel API includes the following:

Create channel.

- Look for the channel that meets some search criteria.
- Modify existing channel.
- Destroying the channel.

Channel program is specified with a PacketProcessingGraph. Composition the PacketProcessingGraph from the PacketProcessingElements allows the reuse of the existing algorithmic blocks and provides flexible and efficient way for describing various packet-processing algorithms.

PacketProcessingGraph architecture is based on the Click Modular Router architecture. It's centered on the *element*. Each element is a software component representing a unit of packet processing. Elements perform conceptually simple computations, such as decrementing an IP packet's time-to-live field, rather than large, complex computations, such as IP routing. They generally examine or modify packets in some way. At run time, elements pass packets to one another over links called *connections*. Each connection represents a possible path for packet transfer. PacketProccessingGraphs are directed graphs of elements with connections as the edges.

Another idea behind channel abstraction is the possibility to share some packet flow (i.e. to process the same packet flow) among several clients (active applications), or contrary the preventing using the channel by others.

Design Considerations

Key factors for the CVM Channel implementation are:

- Flexibility. The user should be able to modify packet-processing scheme for the channel in the convinient manner.
- Modularity. The user should be able to construct various packet-processing algorithms from the small packet processing units – elements.
- Extendibility. The user should be able to make new elements to perform custom packet processing, which is not provided by the default elements.
- Simplicity. The Channel concept should be simple to avoid semantic overloading of the abstraction, to avoid misunderstanding and misusing.
- Expressive power. It should be possible to create complex packet processing algorithms.
- Cooperation. It should be possible to process the same packet flow by different active applications.
- Protection. It should be possible to prevent using packet flow by other clients if required.
- Element Notification Events. Element can notify interested AA about different events.

Functional Structure

All Channel API classes and functions are defined in the cvm::channel namespace.

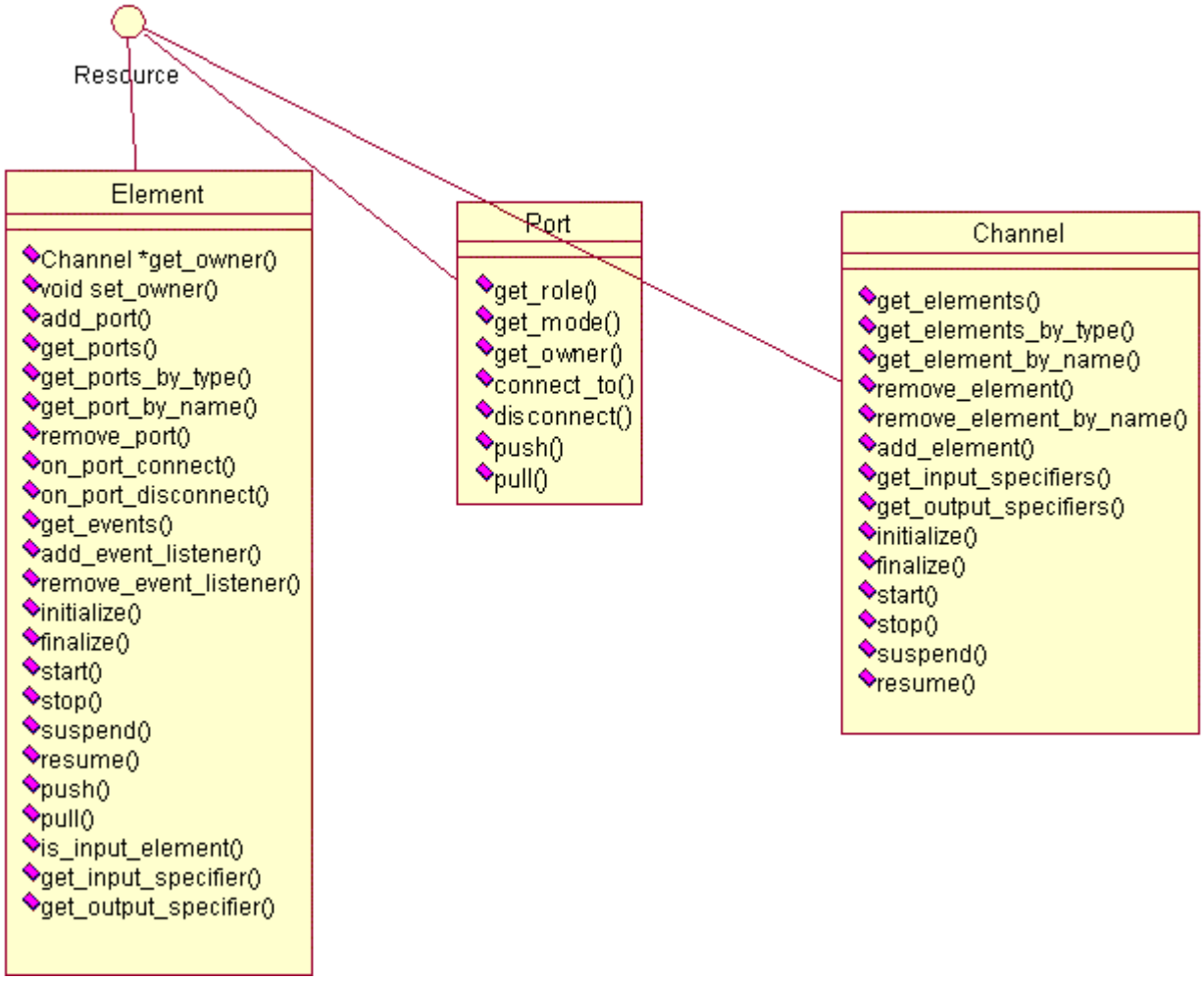
The ChannleAPI consists of the following components:

Interfaces:

- Channel
- InputSpecifier

- OutputSpecifier
- Element
- Port
- Event
- EventListener

Class diagram for the channel API:



- **Channel**

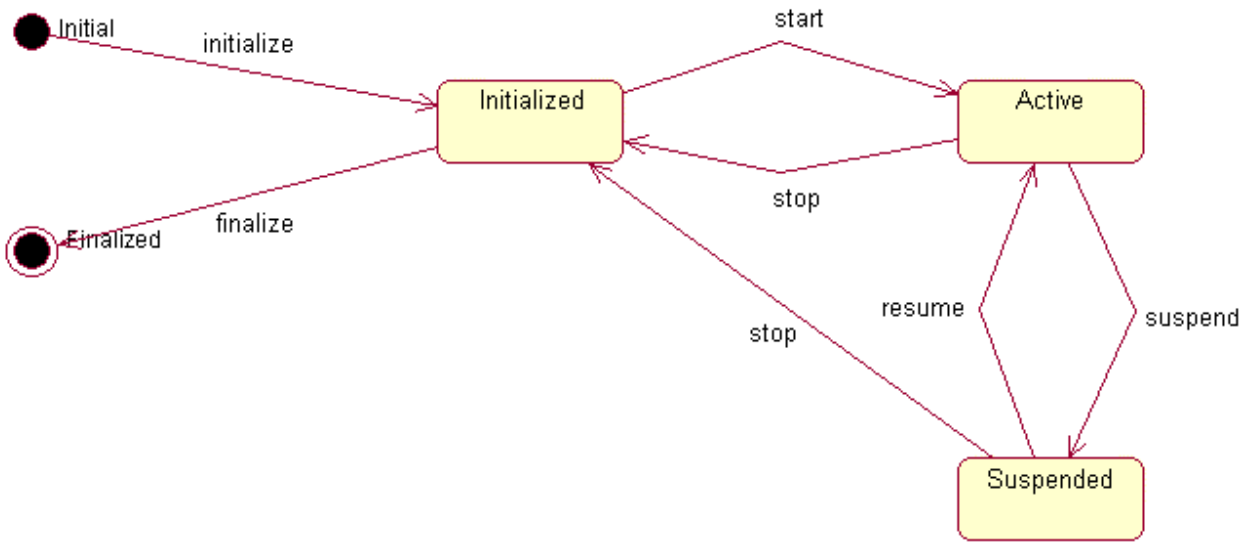
Dk: General comments.
a)Channel lifecycle description and API are missing.
b) Channel validation description and API are missing.

Channel is the packet-processing program. It can be created, activated, deactivated, modified and destroyed.

Typical lifecycle of the channel is

- The Active Application creates the channel.
- The Active Application initializes the channel
- The Active Application initializes the channel
- The Active Application starts the channel to begin packet processing.
- The Active Application can suspend channel and then perform some modifications – add /remove elements, connect/disconnect ports.
- The Active Application cant resume suspended channel to continue packet processing.
- The Active Application can stop channel to finish packet processing.
- The Active Application finalizes channel in order to free resources allocated by the channel.

Here is the channel state chart diagram:



Initialized state differs from the suspended in the internal channel and elements state. Suspend preserves the internal element's state, while stop resets it's to initial.

State transitions happen as the result of the invocation of the channel functions – initialize, start, stop, suspend, resume, finalize.

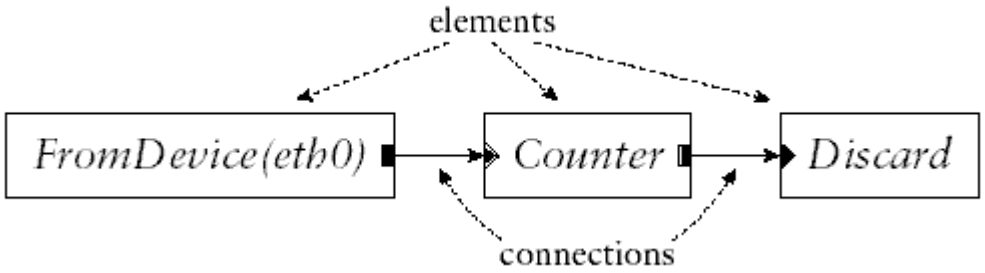
Every channel has:

- Status, which defines its state – active or inactive.
- Access Control List (ACL), which defines the possibility to use the channel by applications other then owner and rules that defines possible actions. Concrete ACL syntax and semantic will be described later.

1. Packet-processing program

Packet-processing praph is constructed from the Elements. Each element is a software component representing a unit of packet processing. Elements perform conceptually simple computations, such as decrementing an IP packet's time-to-live field, rather than large, complex computations, such as IP routing. They generally examine or modify packets in some way. At run time, elements pass packets to one another over links called connections. Each connection represents a possible path for packet transfer. Every connection links an output port on one element to an input port on another. Connections can link ports if the type of the one port matches the type of the other port. The type of the packets that port can receive or emit defines type of the port. Packet- processing programs are directed graphs of elements with connections as the edges.

Figure below shows a simple example of the packet-processing program.



Channel program can be defined with some declarative domain specific language to be specified later. It's possible to modify existing program at runtime using ChannelAPI functions.

2. Element

Element is the software component representing a unit of packet processing. Elements perform conceptually simple computations. It allows run-time introspection. Element has four important properties: element type, ports, events and attributes.

- **Element type.** An element's class specifies element's behavior. Element types support inheritance.

  The ElementType interface represents required information at the runtime. It has at least the name attribute and function returning list of the parent types.

  Dk: Restrictions for element port types are part of element type too. In primitive case there may be no restrictions. Children should inherit these restrictions. For example, base element type has restriction (constraint) that element can have **no more** then 2 ports of type Foo and **at least** 1 port of type Bar. All derived types will inherit constraints for ports of type Foo & Bar, but may in turn define constraints for other port types.

- **Ports.** Each element can have any number of input and output ports. Every connection links an output port on one element to an input port on another. Different ports may serve different purposes; for example, many elements emit some packets on their first output port and other packets on their second. The number of ports provided by an element may be fixed, or it may depend on how many ports were used by the configuration. Every port that is provided must be used by at least one connection, or the configuration is in error. Ports may be push or pull ; these will be defined later.

Dk: Does 'role' forms a part of port type or not ? How 'roles' can be specified formally ?

- Notification Events. Element can produce notification events and notify subscribed Active Applications.

Dk: There must be support for notification events inside element itself.

- Element attributes. Element can have number of attributes. It's possible to set/get values of the attributes using setter/getter functions.

Dk: Element type should define element attributes as well. Thus derived elements should have the same attributes defined in base element type. Children element types may define other attributes as well.

---

3. Port

Port is the point from which the element gets /emits the packets. It can be input or output; push or pull. Element's ports can be inspected and added in the runtime. Port has type and name. Port types can be inherited. The Type interface represents this information at the runtime.

Input and output specifiers represent information about the packet source/destination. Element can be input_element or output_element. In this case it returns the Input or Output specifier which describe where are the packets come from or where do they go to. Channel can give the lists of the input/output specifiers constructed from the specifiers received from the input/output elements.

---

4. InputSpecifier

InputSpecifier is used to represent the information about packet source, which it comes from. InputSpecifier gives the following information: source address, source mask, source port, destination address, destination mask, destination port, protocol and network interface.

---

5. OutputSpecifier

OutputSpecifier describes where the packets go to.

Dk: Does InputSpecifier and OutputSpecifier relate to channel, element or what ? If these are channel attributes then how do they look like?

- **Event**

Element can produce events for subscribed Active Applications. Event has string name and associated pointer to event specific data.

Dk: Events should be sent also "inside elements" to be consumed by element processing code. This code is not a part of AA, it is a part of Element itself. This allows to keep this code in the element even after AA terminates and AA has gone away. Then we can have "persistent channels" that can be created by some AA and stay in the system longer then AA itself.

It makes sense to set aside element processing code or element algorithms in a separate hierarchy of types. Such as "***element algorithm*** that directs packet flow from input port of type Foo to output port of type Bar when it gets element event saying that port of type Bar was added to this element".

- **EventListener**

  EventListener interface used to inform Application about events. Application interested in some events should register EventListener in the Element.

  Dk: This should be called ElementEventListener interface. Element Algorithm types (see comment above) should also implement ElementEventListener interface

---

- **Channel**

```
virtual const std::list<Element *> get_elements() const = 0;
```

Returns list of the channel's elements.

```
virtual const std::list<Element *> get_elements_by_type(const std::string&
                                                         type) const = 0;
```

Returns list of the channel's elements that is a kind of the type.

```
virtual const std::list<Element *> get_elements_by_type(const Type * type)
  const = 0;
```

Returns list of the channel's elements that is a kind of the type.

```
virtual Element * get_element_by_name(const std::string&) const = 0;
```

Returns list of the channel's elements which names are equal to the given argument.

```
virtual bool remove_element(Element * e) = 0;
```

Removes the element from the channel.

```
virtual bool remove_element_by_name(const std::string & element_name) = 0;
```

Removes the elements which names are equal to element_name.

```
virtual bool add_element(Element * e) = 0;
```

Adds element to the channel. . Dk: Where exactly this element is added ?

```
virtual const std::list<InputSpecifier> get_input_specifiers() const = 0;
```

Returns the list of the channel's input specifiers.

```
virtual const std::list<OutputSpecifier> get_output_specifiers() const = 0;
```

Returns the list of the channel's output specifiers.

```
virtual void initialize() = 0;
```

Performs channel initialization. Should be called once before using the channel. Calls initialize function of all channel elements.

```
virtual void finalize() = 0;
```

Performs channel finalization. Should be called once after using the channel. Calls finalize function of all channel elements.

```
virtual void start() = 0;
```

Starts the channel. Calls start function of all channel elements.

```
virtual void stop() = 0;
```

Stops the channel. Calls stop function of all channel elements.

```
virtual void suspend() = 0;
```

Suspends the channel. Calls suspend function of all channel elements.

```
virtual void resume() = 0;
```

Resumes the channel suspended by the call of the ‘suspend’ function. Calls ‘resume‘ function of all channel elements. Active application can call suspend & resume to modify existing active channel.

Dk: When we ‘suspend’ and ‘resume’ channel ?

- **Element**

```
virtual Channel * get_owner() const = 0;
```

Returns the channel that element belongs to.

```
virtual void set_owner(Channel * owner) = 0;
```

Sets the element’s owner.

```
virtual Port * add_port(Port * port) = 0;
```

Adds port to the channel.

```
virtual Port * add_port(const std::string & name,
                        const std::string & type,
                        Port::PortRole role,
                        Port::PortMode mode) = 0;
```

Adds port to the channel.

```
virtual Port * add_port(const std::string & name,
                         const Type * type,
                         Port::PortRole role,
                         Port::PortMode mode) = 0;
```

Adds port to the channel.

```
virtual std::list<Port *> get_ports() const = 0;
```

Returns the list of all element’s ports.

```
virtual std::list<Port *> get_ports_by_type(const std::string& type) const=0;
```

Returns the list of the element’s ports that conforms to the type.

```
virtual std::list<Port *> get_ports_by_type(const Type * type) const = 0;
```

Returns the list of the element’s ports that conforms to the type.

```
virtual Port * get_port_by_name(const std::string&) const = 0;
```

Returns the list of the element’s ports which names are equal to the given argument.

```
virtual bool remove_port(const Port * p) = 0;
```

Removes port from the element.

```
virtual bool remove_port(const std::string & p)
```

Removes port from the element.

```
virtual void on_port_connect(Port * out_port, Port * in_port) = 0;
```

Called when port is connected.

Dk: ‘Element AlgorithmS’ should implement this interface.

```
virtual void on_port_disconnect(Port * out_port, Port * in_port) = 0;
```

Called when port is disconnected.

```
virtual std::list<std::string> get_events() const = 0;
```

Returns the list of the events available to listen to.

```
virtual int add_event_listener(const std::string & event_name,
                               EventListener * listener) = 0;
```

Returns the id of the event listener registered to listen for the event.

```
virtual int remove_event_listener(int listener_id) = 0;
```

Removes the event listener.

```
virtual void initialize() = 0;
```

Performs element initialization. Called by the channel at the channel initialization phase.

```
virtual void finalize() = 0;
```

Performs element finalization. Called by the channel at the channel finalization phase.

```
virtual void start() = 0;
```

Starts the element. Called by the channel at the channel start phase.

```
virtual void stop() = 0;
```

Sops the element. Called by the channel at the channel stop phase.

```
virtual void suspend() = 0;
```

Suspends the element. Called by the channel at the channel suspend phase.

```
virtual void resume() = 0;
```

Resumes the element. Called by the channel at the channel resume phase.

```
virtual void push(Packet * packet, Port * port) = 0;
```

Called when packet ‘packet’ is arrived from the port ‘port’.

```
virtual Packet * pull(Port * port) = 0;
```

Called when packet is requested from the port ‘port’.

```
virtual bool is_input_element() const;
```

Returns true if the element is the input element. In such case it should return non-empty input specifier.

```
virtual bool is_output_element() const;
```

Returns true if the element is the input element. In such case it should return non-empty output specifier.

```
virtual InputSpecifier get_input_specifier() const;
```

Returns input specifier.

```
virtual OutputSpecifier get_output_specifier() const;
```

Returns output specifier.

Dk: Define InputSpecifier and OutputSpecifier !!!

- **Port**

```
enum PortRole {INPUT_PORT, OUTPUT_PORT};
enum PortMode {PUSH_MODE, POOL_MODE};
```

```
virtual PortRole get_role() const = 0;
```
Returns role of the port.

```
virtual PortMode get_mode() const = 0;
```
Returns mode of the port.

```
virtual Element * get_owner() const = 0;
```
Returns owner of the port.

```
virtual bool connect_to(Port * other) = 0;
```
Connects to other port.

```
virtual bool disconnect(Port * other) = 0;
```
Disconnects from the other port.

```
virtual void push(Packet * p) = 0;
```
Transfers the packet to the connected port or to the port owner.

```
virtual Packet * pull() = 0;
```
Requests the packet from the connected port or the port owner.

- **InputSpecifier**

```
uint32_t src_addr;
uint32_t src_mask;
uint16_t src_port;
uint32_t dst_addr;
uint32_t dst_mask;
uint16_t dst_port;
uint16_t protocol;
std::string network_interface;
```

- **OutputSpecifier**

```
uint32_t src_addr;
uint32_t src_mask;
uint16_t src_port;
uint32_t dst_addr;
uint32_t dst_mask;
uint16_t dst_port;
uint16_t protocol;
std::string network_interface;
```

- **Event**

Dk: ElementEvent, ElementEventListener

virtual const std::string & get_name() const = 0;
Returns the name of the event.

virtual void * get_data() const = 0;
Returns the data of the event.

- **EventListener**

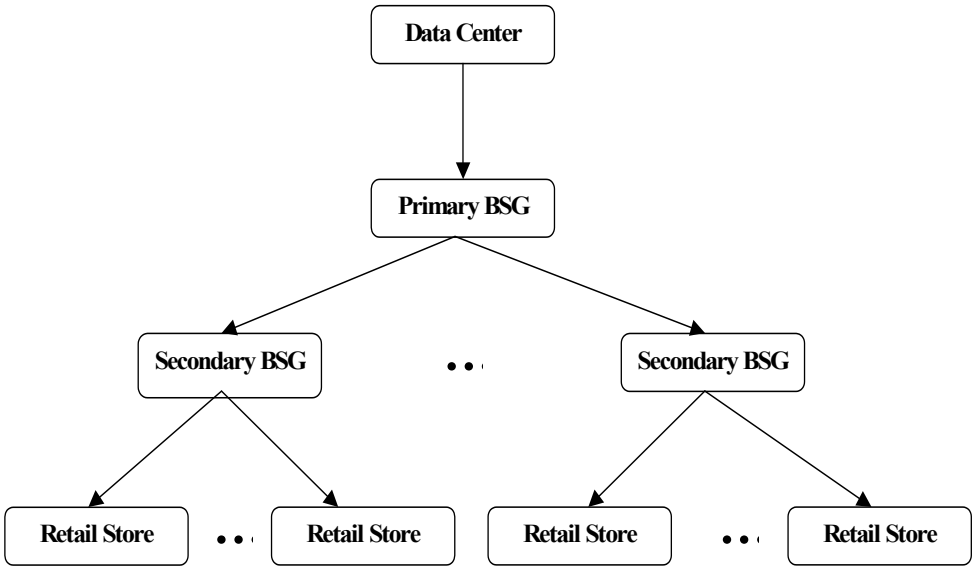
```
virtual void process_event(const Event * event) = 0;
```

Processes the event.

ATTACHMENTS

- **Example of using CVM Channels for simple TFTP multicasting implementation.**

  In this section we demonstrate using the CVM Channels for simplest version of the TFTP - ARM demonstration. Network configuration is shown on the picture below.



In our test implementation we assume that there are the following entities:

- Data Center
- Primary BSG
- One or more secondary BSGs
- One or more Retail Stores

Data Center uses unicast link to connect to the Primary BSG. Primary BSG acts like simple TFTP server with the following restrictions:

- It supports only Write Requests
- It implements a standard unicast TFTP

At the primary BSG side there are the following active applications:

1. TFTP server. It listens for TFTP write requests and establishes the session to receive the file.
2. ARM File Transfer sender. It multicasts the file to the specified multicast group.

Below is the simple file transfer scenario at the primary BSG:

1. TFTP server application creates the TFTP server channel.

2. TFTP server channel listens on the TFTP port (69).
3. Data Center sends Write Request to the BSG TFTP port.
4. TFTP server acquires available UDP port.
5. TFTP server creates TFTP receiver channel with the filter tuned to the source address of the incoming packet and the number of the acquired UDP port.
6. TFTP server sends Write Request packet to the TFTP receiver channel.
7. TFTP receiver implements TFTP protocol and passes the incoming data to the application-specific adaptor (TFTP-ARM adaptor in our case).
8. TFTP receiver gets last packet, notifies the adaptor closes channel.
9. TFTP-ARM adaptor stores data in the cache.
10. ARM channel retrieves the data from the cache and transfers the file using the ARM file transfer protocol.
11. ARM channel completes its work.

- **TFTP channels**

6. TFTP server

TFTP server active application uses two different channels to implement the required functionality. The first channel is always active. It listens on the TFTP port (69 by default). When it receives the Write Request, the second channel is initialized and this second channel performs the task of receiving the file. The channels are called Server channel and Server Receiver channel respectively. The separate Server Error channel is used to report TFTP error if the TFTP session cannot be started for some reason.

- **Server channel**

The Packet Processing Graph is shown in Figure 1.

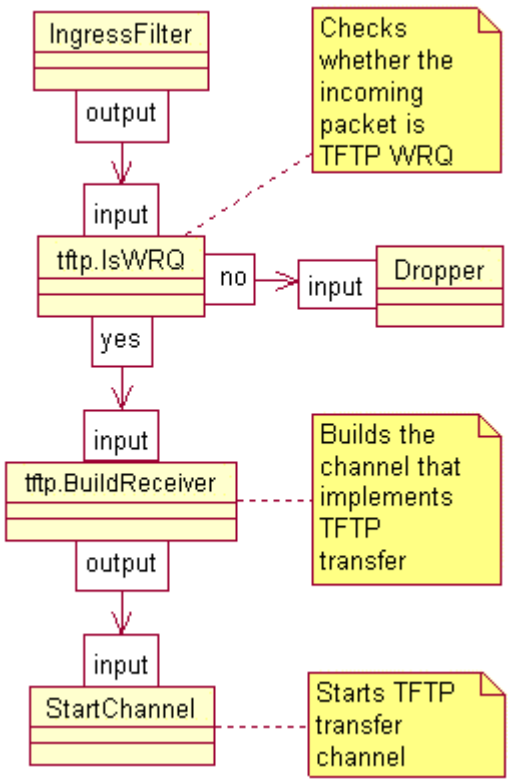


Figure 1. TFTP server channel

Here is a description of the packet traversal in this channel:

1. The packet sent to TFTP port (69 by default) is caught by the filter.
2. The Condition element IsWRQ checks whether the packet is WRQ.
3. If no, the packet is dropped.
4. If yes, the packet causes BuildReceiver to allocate the unused UDP port number, to build the TFTP Server Receiver channel using this port and to set the packet destination port to the allocated value.
5. The StartChannel element starts the constructed Server Receiver channel and sends the WRQ packet to it.

- ## Server Receiver channel

The element BuildReceiver creates the TFTP receiver channel as described in the section above. The corresponding Packet Processing Graph is shown in Figure 2.

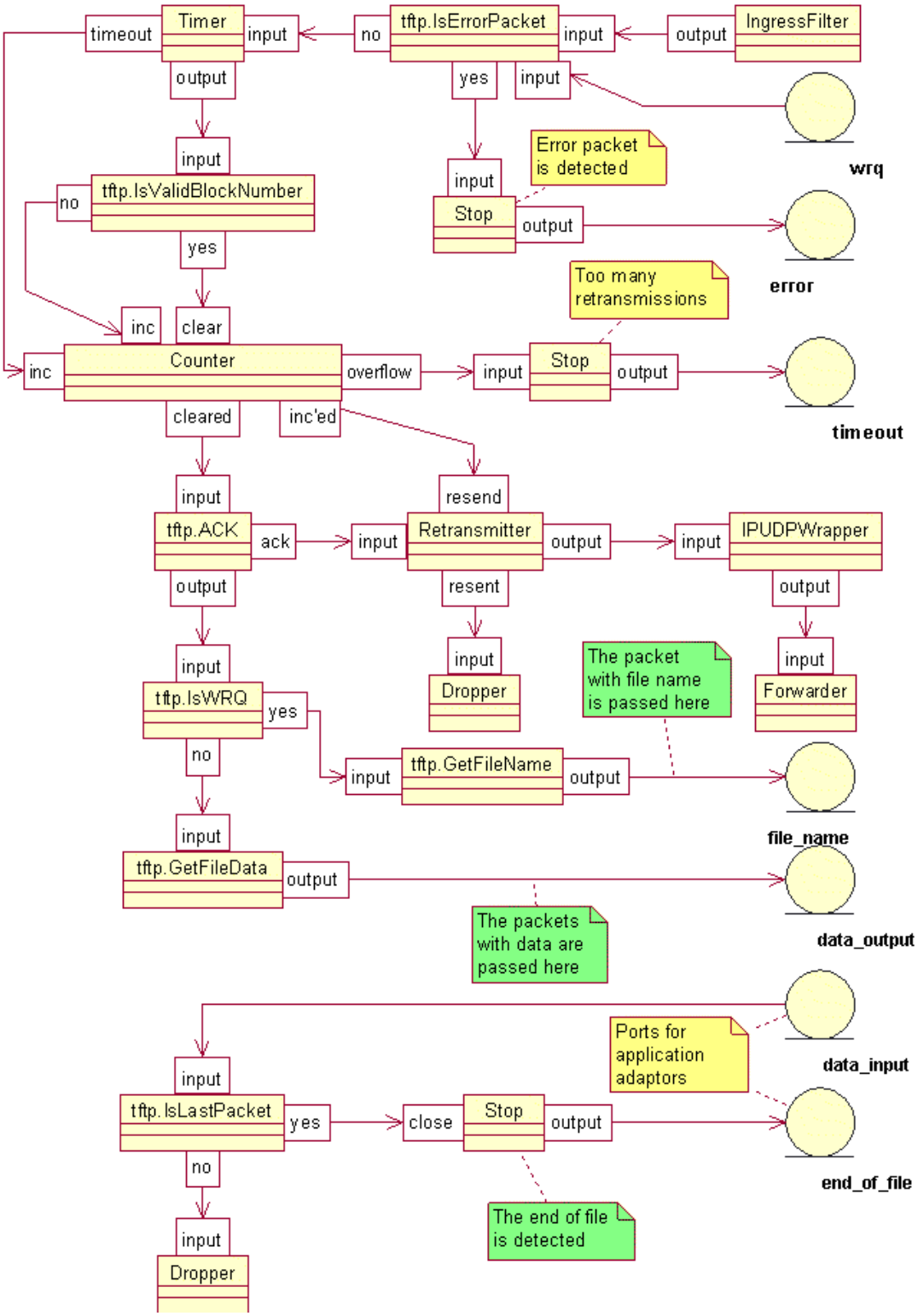


Figure 2. TFTP server receiver channel

The packet traverses the channel as follows:

1. The incoming packet is checked whether it is an error packet.

2. If so, the counterpart is notified via error port and the channel is stopped.
3. Otherwise, the packet resets the Timer.
4. The packet is checked whether the block number is correct.
5. Whenever the block number is incorrect or the timer expires, the retransmission Counter is incremented.
6. If the increment causes the Counter to overflow, the channel is stopped due to excessive retransmissions.
7. Otherwise the increment packet forces the Retransmitter to retransmit the recently sent packet.
8. The packet that cleared the Counter is sent to ACK element.
9. This element creates the corresponding acknowledgement packet.
10. The acknowledgement packet is passed to the Retransmitter to remember it for future retransmissions and then is forwarded to the network.
11. The original packet is checked whether it is a WRQ packet.
12. If so, the file name is extracted from the packet and passed to the counterpart.
13. Otherwise, the file data is extracted from the packet and is supplied to the counterpart via data_output port.
14. The counterpart passed the data packet back via the data_input port.
15. The packet is then tested whether it is the last packet in the sequence.
16. If no, it is dropped.
17. Otherwise the packet causes the counterpart to be notified and the channel to stop.

- **Server Error channel**

The channel is used to report the error to the client when the server cannot process the incoming request because of some application-specific reason, e.g. when there are no free ARMTP sessions available.

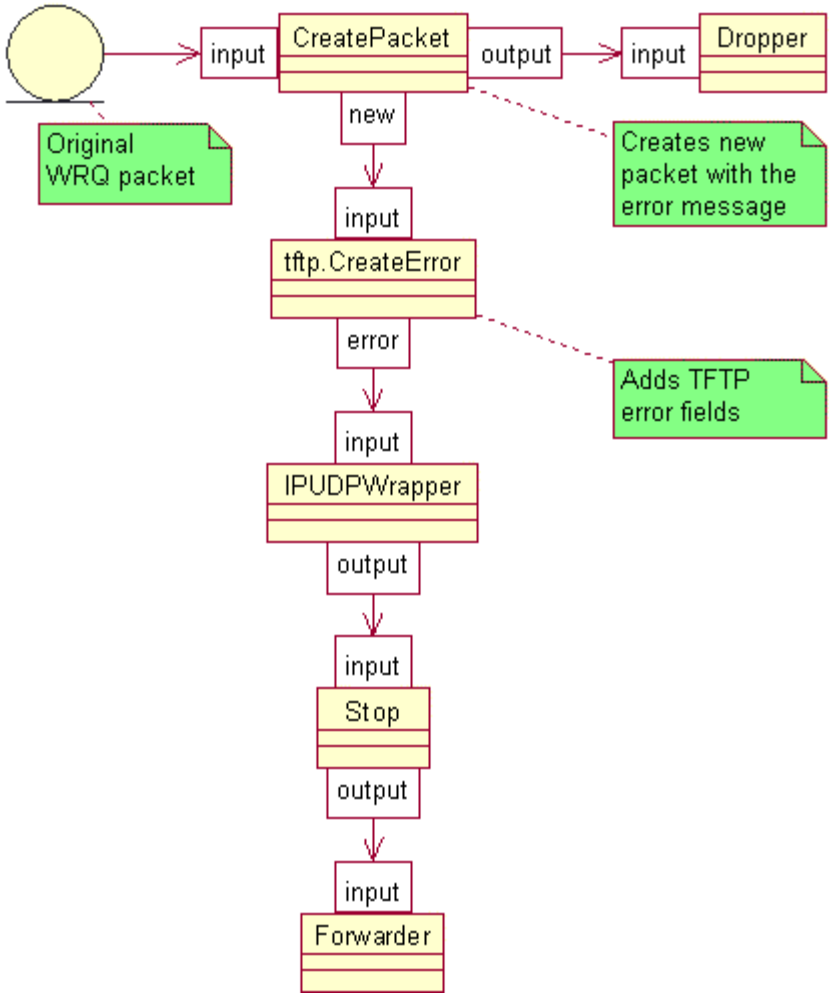


Figure 3. TFTP server error channel

---

7. TFTP client (sender)

After the file arrives to the edge BSG, it should be transmitted to the client machines using TFTP as well. So, the edge BSG runs TFTP sender applications that perform this task. The TFTP sender appears to be a TFTP client that sends the Write Request to the Retail Store. The Packet Processing Graph for TFTP sender channel is shown in Figure 4.

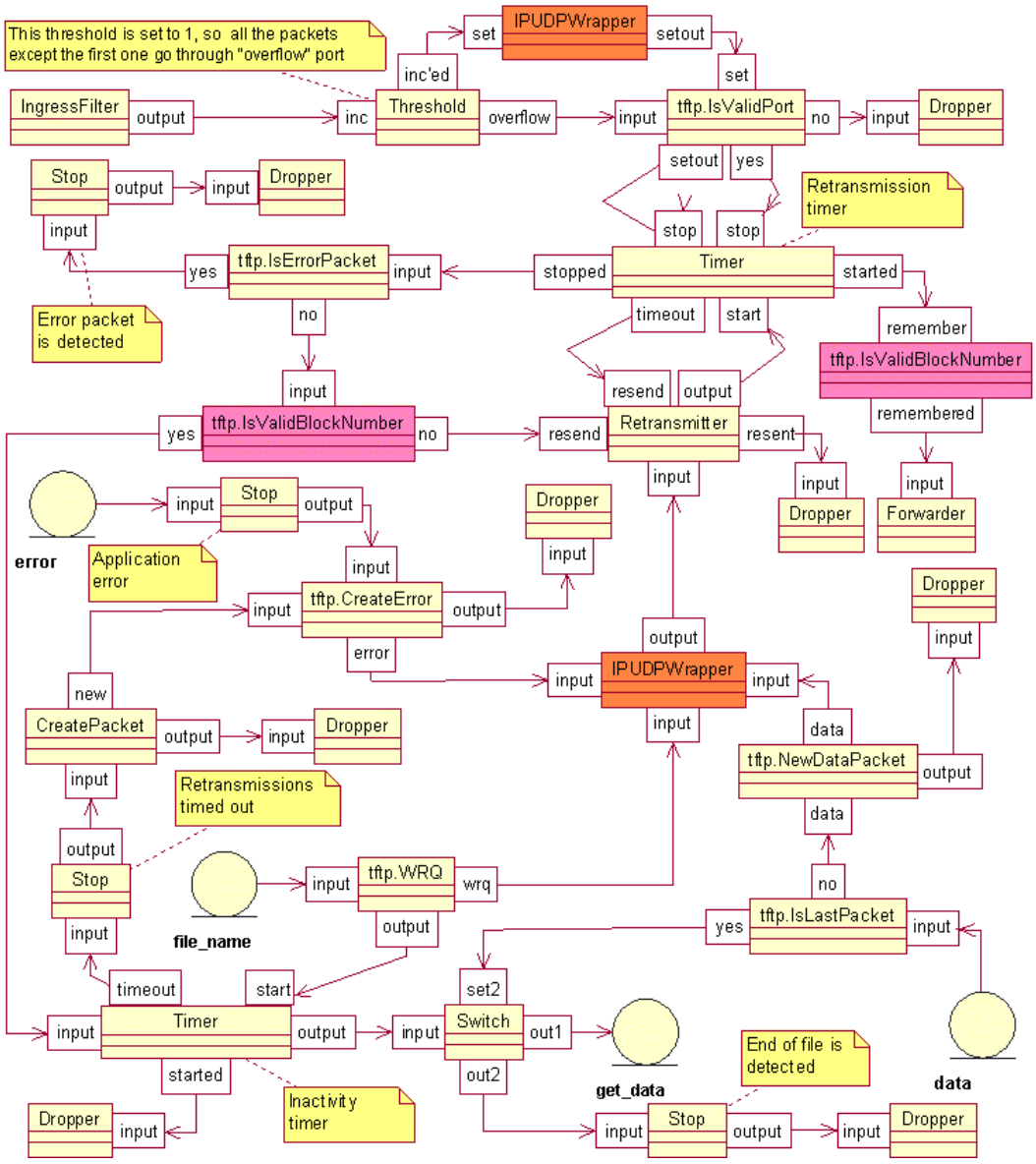


Figure 4. TFTP sender channel

The packets traverse the channel as follows:

1. The first packet is generated by the WRQ element.
2. This packet starts the inactivity Timer and is passed to the Retransmitter.
3. Retransmitter remembers the packet.
4. The retransmission Timer is started.
5. The block number of the packet being sent is remembered by the IsValidBlockNumber element.
6. The packet is forwarded to the network.
7. The incoming packets are selected by the IngressFilter.

8. The first incoming packet is selected from all the other packets by the Threshold element.
9. It is used to set the destination port of the data packets (element IPUDPWrapper).
10. Then this packet is used to remember the port number in the Condition element IsValidPort.
11. The second and next packets are first checked whether they have the same port number as the first packet has.
12. If no, the packets are dropped.
13. Otherwise the packet stops the retransmission Timer.
14. The packet is checked whether it is an error packet and if yes, the channel is stopped.
15. The packet is checked whether they have the valid block number.
16. If yes, the packet resets inactivity Timer.
17. When the inactivity Timer expires, the error packet is sent to the client and the channel is stopped due to excessive retransmissions.
18. Alternatively the session can be terminated from the outside via error port.
19. The packets that reset inactivity Timer are passed to the Switch element and normally go to the out1 port and to the get_data counterpart port.
20. If the last packet is transmitted the Switch element passes the packet to out2 port and the channel is stopped.
21. The counterpart passes the data via data port to the NewDataPacket element.
22. The new data packet is created by the NewDataPacket element.
23. Then the packet obtains the proper IP and UDP headers in IPUDPWrapper element.
24. The packet is then sent throught the Retransmitter to the Forwarder.

- **ARMTP channels**

This section describes more real channels for ARMTP.

For clarity the following convention is used on the diagrams:

- The port connection ends are not marked with the port names for ports named “input” and “output”
- Asterisk (*) is used to denote all output element ports not explicitly shown on the diagram
- As usual the elements that appear on the diagram twice or more with the same specific color denote one element

8. Data receiver

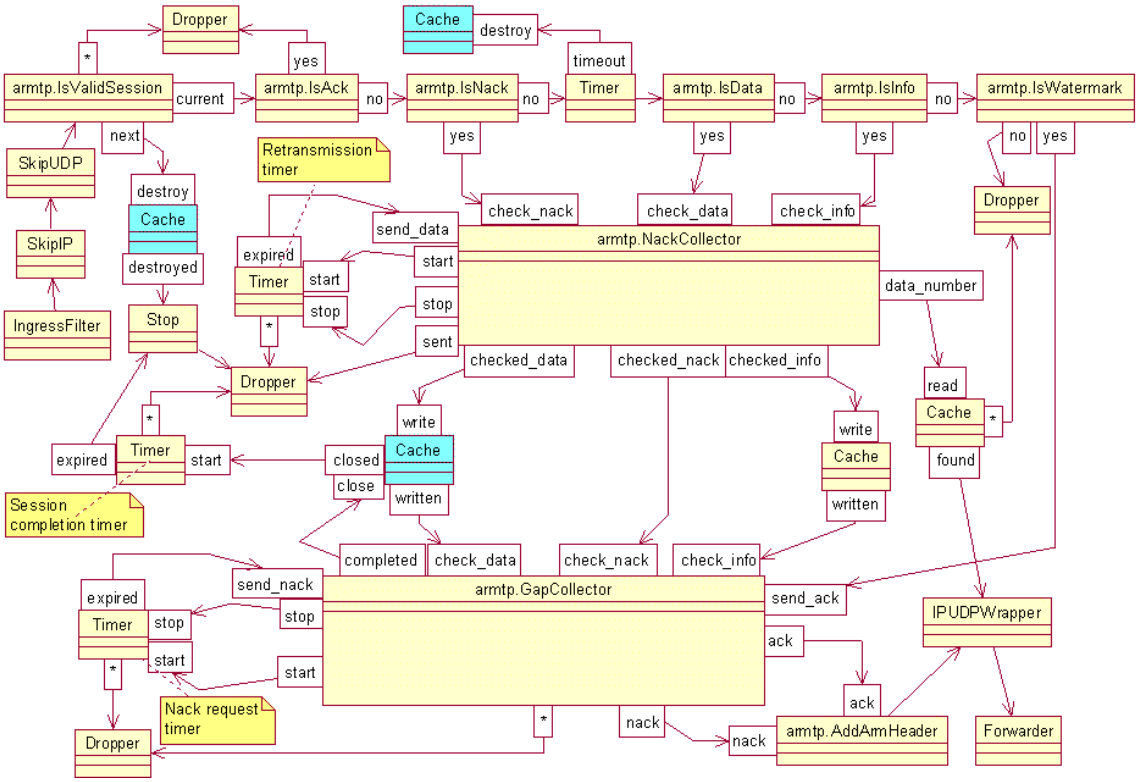


Figure 5. ARMTP receiver channel

9. Data sender

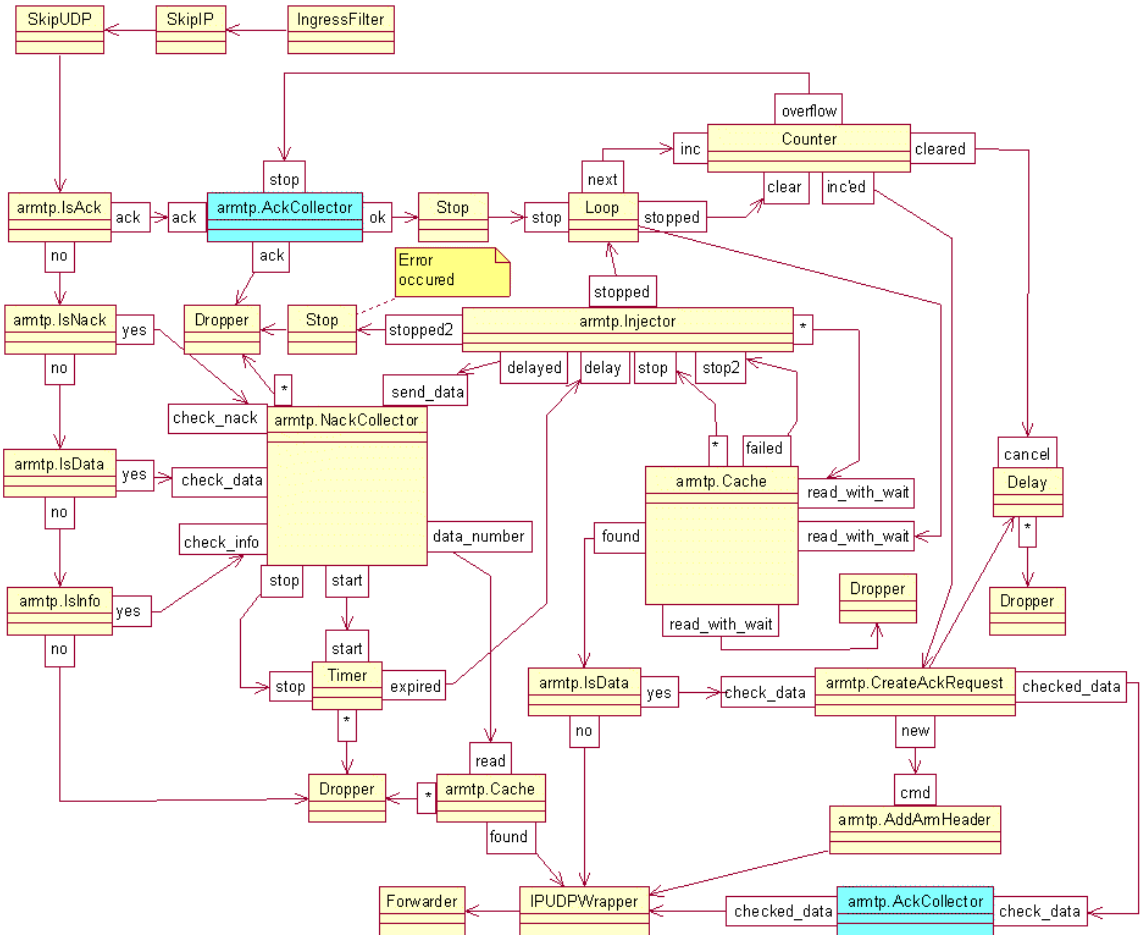


Figure 6. ARMTP sender channel

- **TFTP and ARM adaptors**

10. ARM Receiver adaptors

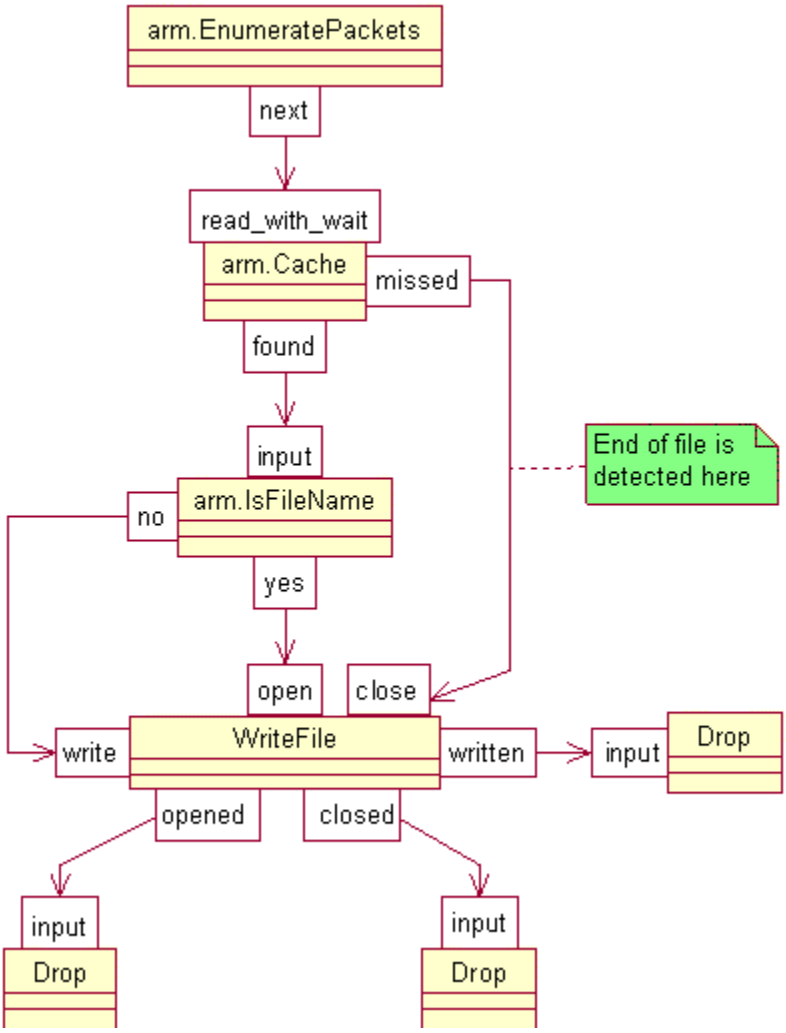


Figure 7. Receiver adapter: saving to file

11. ARM Sender adaptors

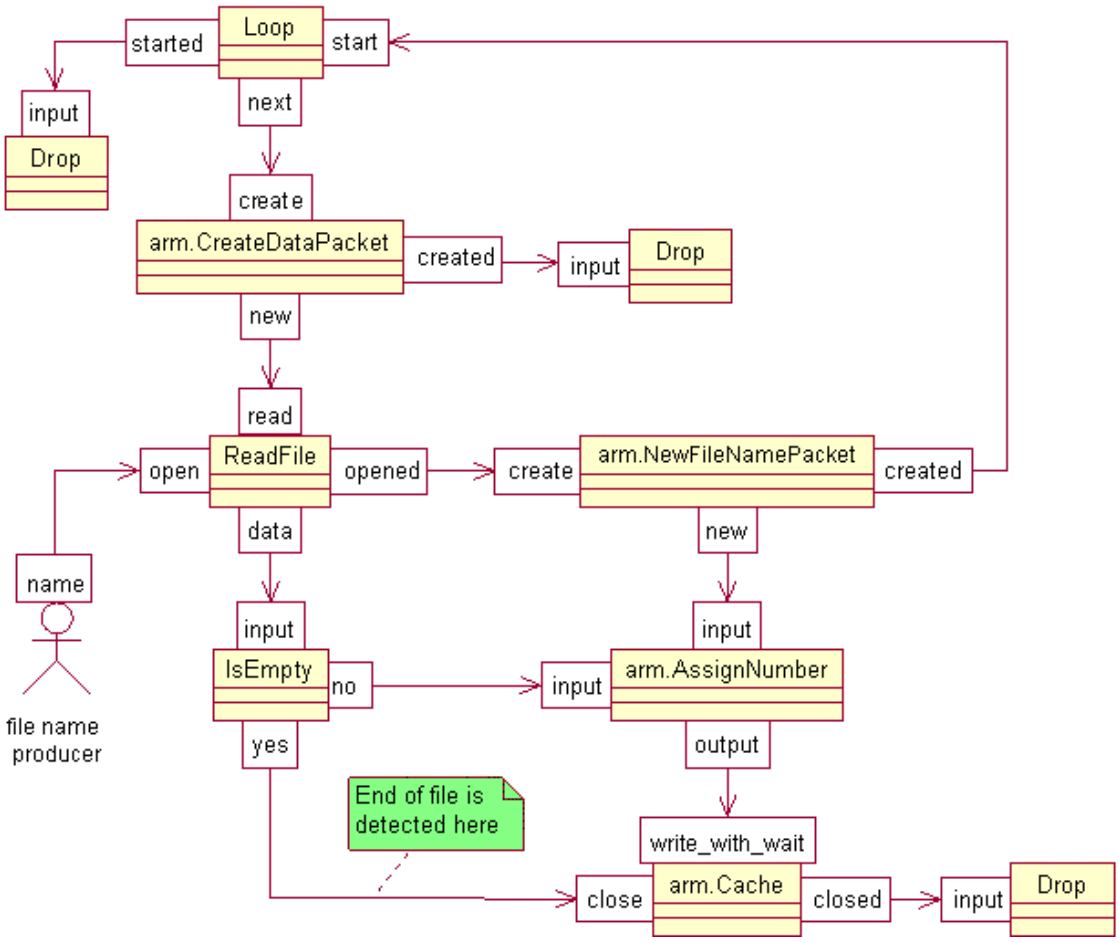


Figure 8. Sender adapter: reading from file

- **Standard Elements**

The channel framework provides a set of elements ready for use in the active applications. They are described in this section.

12. IngressFilter

Outputs the packets corresponding to the given ingress filter.

Ports

`output`	1	PUSH OUTPUT	the port to that the packets are output

Semantics

Sends the packets that match the ingress filter to the `output` port.

---

13. Dropper

Drops the incoming packets.

Ports

`input` 0..n PUSH INPUT the port for incoming packets

Semantics

Drops the packets that arrive to the `input` port.

---

14. Forwarder

Sends the given packet to the network.

Ports

`input` 0..n PUSH INPUT the port for incoming packets

Semantics

Sends the packets that arrive to the `input` port to the network.

---

15. Reclassifier

Sends the given packet back to the Packet Manager for reclassification.

Ports

`input` 0..n PUSH INPUT the port for incoming packets

Semantics

Sends the packets that arrive to the `input` port back to the Packet Manager for reclassification.

---

16. Stop

Stops the channel.

Ports

`input`	0..n	PUSH INPUT	the port for incoming packets
`output`	1	PUSH OUTPUT	the port for outgoing packets

Semantics

Any packet that arrives on the `input` port is passed to the `output` port without any modification. When the transmittion is completed, the channel with this element is stopped.

---

17. Retransmitter

Retransmits the previously sent packet.

Ports

`input`	0..n	PUSH INPUT	the port for incoming packets that may be retransmitted later
`output`	1	PUSH OUTPUT	the port for packets that are transmitted or retransmitted
`resend`	0..n	PUSH INPUT	the port for incoming packets that trigger retransmission
`resent`	1	PUSH OUTPUT	the port for packets that triggered retransmission

Semantics

Any packet passed to the `input` port is remembered by the element and is passed without any modification to the `output` port. Any packet passed to the `resend` port causes the last remembered packet to be retransmitted to the `output` port and then the initial packet is sent to the `resent` port.

---

18. Timer

Generates the timeout packets if there are no incoming packets within the specified time interval.

Ports

`input`	0..n	PUSH INPUT	the port for incoming packets
`output`	1	PUSH OUTPUT	the port for outgoing packets
`timeout`	1	PUSH OUTPUT	the port to which the timeout packets are sent

Semantics

Any packet passed to the `input` port is resets the timer and is transmitted to the `output` port without any modification. If there are no packets passed to the `input` port within the specified time interval, the new packet is sent to the `timeout` port.

19. Counter

Counts the incoming packets and passes them to the one of the two output ports depending whether the maximal count is reached or not.

Ports

`inc`	0..n	PUSH INPUT	the port for incoming packets to be counted
`inc'ed`	1	PUSH OUTPUT	the port for packets which numbers do not exceed the specified amount
`overflow`	1	PUSH OUTPUT	the port for packets which numbers exceed the specified amount
`clear`	0..n	PUSH INPUT	the port for incoming packets that clear the number of packets
`cleared`	1	PUSH OUTPUT	the port for packets that cleared the number of packets

Semantics

Any packet passed to the `inc` port causes the internal count to be incremented by 1. If the counter is within the specified range, the packet is passed unmodified to the `inc'ed` port, otherwise the packet is passed to the `overflow` port. Any packet passed to the `clear` port sets the internal counter to 0 and the packet itself is sent unmodified to the `cleared` port.

---

20. Threshold

Counts the incoming packets and passes them to the one of the two output ports depending whether the maximal count is reached or not.

Ports

`inc`	0..n	PUSH INPUT	the port for incoming packets to be counted
`inc'ed`	1	PUSH OUTPUT	the port for packets which numbers do not exceed the specified amount
`overflow`	1	PUSH OUTPUT	the port for packets which numbers exceed the specified amount

Semantics

Any packet passed to the `inc` port causes the internal count to be incremented by 1. If the counter is within the specified range, the packet is passed unmodified to the `inc'ed` port, otherwise the packet is passed to the `overflow` port.

---

21. Switch

Passes the incoming packets to one of two output ports depending on the element state.

Ports

`input`	0..n	PUSH INPUT	the port for incoming packets
`output1`	1	PUSH OUTPUT	the port 1 for the outgoing packets
`output2`	1	PUSH OUTPUT	the port 2 for the outgoing packets
`set1`	0..n	PUSH INPUT	the port for packets that redirect the incoming packets to port 1
`done1`	1	PUSH OUTPUT	the port for packets that redirected the incoming packets to port 1
`set2`	0..n	PUSH INPUT	the port for packets that redirect the incoming packets to port 2
`done2`	1	PUSH OUTPUT	the port for packets that redirected the incoming packets to port 2

Semantics

Any packet from the `input` port is passed initially to `output1` port. The packet from the `set2` port causes these packets to be routed to the `output2` port and the packet itself leaves from the `done2` port. The pair of the `set1/done1` port works the same way but for the `output1` port.

---

22. Condition

Passes the incoming packets to one of two output ports depending on the packet contents. This is a pseudo-element that defines the ports and the semantics, but the actual condition is defined in the specific elements.

Ports

`input`	0..n	PUSH INPUT	the port for incoming packets
`yes`	1	PUSH OUTPUT	the port for the packets for which the condition is true
`no`	1	PUSH OUTPUT	the port for the packets for which the condition is false

Semantics

Any packet from `input` port is passed unchanged to the output port `yes` if the condition for that packet is true and to the output port `no` otherwise.

---

23. FileReader

Fills the given packet with the data taken from the file.

Ports

`input`	0..n	PUSH INPUT	the port for the incoming packets
`output`	1	PUSH OUTPUT	the port for the packets filled with the data

Semantics

Any packet from the `input` port is filled with the data taken from the file (the file is specified at the element initialization stage) and is passed to the `output` port. If the file contains more data than the packet data size only the packet data size is read from the file and is written to the packet. The next packets will contain the next file data. If the file data size is less than the packet data size, the packet data size is reduced to the size of the file data.

---

24. WriteFile

Stores the given packet data to the file.

Ports

`input`	0..n	PUSH INPUT	the port for the incoming packets that contains the data
`output`	1	PUSH OUTPUT	the port for the packets which data was written to the file
`open`	0..n	PUSH INPUT	the port for the incoming packets that open the file
`opened`	1	PUSH OUTPUT	the port for the packets that opened the file
`close`	0..n	PUSH INPUT	the port for the incoming packets that close the file
`closed`	1	PUSH OUTPUT	the port for the packets that closed the file

Semantics

The data of any packet from the `input` port is written to the previously open file and is passed to the `output` port. The packets sent to `open` port cause the file to be open. The name of the file is taken from this packet. The packets sent to `close` port cause the file to be closed.

---

25. StartAA

Starts Active Applications.

Ports

`start`	0..n	PUSH INPUT	the port for the incoming packets that cause the AA to start
`started`	1	PUSH OUTPUT	the port for the packets that caused AA to start
`set_param`	0..n	PUSH INPUT	the port for the incoming packets that contain AA parameter
`done_param`	1	PUSH OUTPUT	the port for the packets that are used to set AA parameter

Semantics

Any packet from the `start` port causes the AA to start. The AA settings like AA name, URL and initial parameter string are specified when the element is created. This packet sent to the `start` port is

then output on the `started` port. The packet sent to the port `set_param` adds the packet data to the initial parameter string. The packet is then output at the `done_param` port.

---

26. IPUDPWrapper

Adds IP and UDP headers to the input packet.

Ports

`input` 0..n PUSH INPUT the port for the incoming packets

`output` 1 PUSH OUTPUT the port for the packets with added IP and UDP headers

`set_dstport` 0..n PUSH INPUT the port for the packets that set the destination UDP port

`done_dstport` 1 PUSH OUTPUT the port for the packets sent to `set_dstport`

Semantics

To any packet from the `input` port the element adds the valid UDP and IP headers. The header fields like IP addresses and UDP ports are specified at the element creation time. Packets sent to the `input` port are then output to the `output` port. The packets sent to the port `set_dstport` modify the UDP destination port number for the outgoing packets and are output from the `done_dstport` port.

- **TFTP specific elements**

27. Condition elements

The elements listed below implement the *Condition* element described earlier. The corresponding condition is specified for every such element.

Element	**Condition**
isWRQ	Is the incoming packet a Write Request (WRQ)?
isErrorPacket	Is the incoming packet an Error packet?
isValidBlockNumber	Is the block number in the packet 1 greater than in the previous packet?
isLastPacket	Is the packet last in the stream (the data size is less than 512)?
isValidPort	Is the UDP port specified in the packet equal to the specified one?

28. BuildReceiver

Builds the receiver channel and activates it.

Ports

`input`	0..n	PUSH INPUT	the port for incoming WRQ packets
`output`	1	PUSH OUTPUT	the port for the outgoing packets

Semantics

Any packet from the `input` port causes the element to build the TFTP receiver channel and to activate it. Then the destination port number in the incoming packet is modified to the port number of the TFTP receiver and the packet is sent to the `output` port.

29. ACK

Produces ACK packet for the given data or request packet.

Ports

`input`	0..n	PUSH INPUT	the port for incoming packets
`output`	1	PUSH OUTPUT	the port for the original packets
`ack`	1	PUSH OUTPUT	the port for the ACK packets

Semantics

For any packet from the `input` port the corresponding ACK packet is sent to the `ack` port while the original packet is passed unchanged to the `output` port.

30. WRQ

Produces the Write Request (WRQ) packet.

Ports

`output`	1	PUSH OUTPUT	the port for WRQ packet

Semantics

When the element is started, it produces the WRQ (Write Request) packet at the `output` port with the contents according to the parameters passed during the element initialization.

31. NewDataPacket

Produces the data packet.

Ports

`input`	0..n	PUSH INPUT	the port for incoming packets
`output`	1	PUSH OUTPUT	the port for the original packets
`data`	1	PUSH OUTPUT	the port for the new data packets

Semantics

For any packet from the `input` port the corresponding new data packet is sent to the `data` port while the original packet goes to the `output` port. The data packet addresses, ports, etc. are set according to the initialization parameters. The data packet has a room for 512 bytes of UDP data. It's a responsibility of the next elements to fill in the actual data and to shrink the packet size when required.

---

32. GetFileName

Extracts the file name from either WRQ or RRQ packets.

Ports

`input`	0..n	PUSH INPUT	the port for incoming WRQ/RRQ packets
`output`	1	PUSH OUTPUT	the port for the outgoing packets with file name as data

Semantics

Any packet from the `input` port is modified so that its data contains the file name. The file name length matches the data length. The packet is then sent to the `output` port.

---

33. GetFileData

Extracts the file data from the TFTP data packets.

Ports

`input`	0..n	PUSH INPUT	the port for incoming data packets
`output`	1	PUSH OUTPUT	the port for the outgoing packets with file data

Semantics

Any packet from the `input` port is modified so that its data contains the file data. The file data length matches the data length. The packet is then sent to the `output` port.

- **Multicast specific elements**

34. Condition elements

The elements listed below implement the *Condition* element described earlier. The corresponding condition is specified for every such element.

Element	Condition
IsValidSessionID	Is the session ID equal to the given one?
IsValidFileID	Is the file ID equal to the given one?
IsValidBlockNumber	Is the block number 1 greater than in the previous packet?
IsLastBlock	Is the packet last for the given file? (The actual definition may vary according to the session parameters, e.g. it may be defined in the way it is done for the TFTP channel)

35. NewDataPacket

Produces the data packet.

Ports

`data` 1 PUSH OUTPUT the port for the new data packets

Semantics

Creates the new data packet and sends it to the `data` port. The data packet addresses, ports, etc. are set according to the initialization parameters. The data packet has a room for 512 bytes of data. It's a responsibility of the next elements to fill in the actual data and to shrink the packet size when required.