1 Overview

The Active Filter Signaling Protocol Version 2 (AFSPv2) is a signaling protocol that transmits control messages upstream (towards data publishers) and downstream (towards subscribers) in active multicast trees. These control messages are responsible for reliably setting up, maintaining and tearing down filter states that are installed at each active router or end host. An AFSPv2 module interacts with three adjacent modules, the Active Topology Discovery Protocol (ATDP) module, the Filter Manager (FM) module and the application (on end host only). Figure 1 illustrates the position of AFSPv2 module in the overall system architecture. Since FM is responsible for maintaining the database of all the related filter request and controlling data filter setup, the goal of AFSPv2 is to accept filter operation request from the application and to ensure that such information reach all the FMs located on the multicast data path. Furthermore, since the data dissemination path may change due to routing changes, AFSPv2 should also provide sufficient information for FM to remove obsolete filter requests associated with the old topology and to setup new filters associated with the new topology.

1.1 Design principles

The current version of AFSPv2 aims at implementing a subscription-based filtering system, where active filters are established only based on the content subscription requests initiated by the receivers. However in a general publish/subscribe system, both subscription requests (filter requests) and publication announcements can be useful for filter setups. We will investigate integrating publication announcements as a future extension.

For better portability, AFSPv2 is designed as a separate module with limited but well-defined interfaces with adjacent modules. Importantly, AFSPv2 does not depend on understanding the content of filter requests, such as the filter variables or the interest domains, to properly operate. Filter requests

*This document is written by Ping Ji, Zihui Ge, and Jonathan Shapiro of Umass at Amherst. Please contact Ping, Zihui or Jonathan at \{jiping, gezihui, jshapiro\}@cs.umass.edu for any questions or comments.
should be opaque to AFSPv2, thus, when filter formats or definitions are changed (e.g., in order to support different application), only FM needs to be modified.

At present, no filter aggregation is allowed at any of the intermediate active nodes. This is due to consideration of three different aspects: (a) signaling latency — performing filter aggregation requires AFSPv2 to first query FM, which would conduct several geometric comparisons, and would then pass back the query result to AFSPv2; since geometric comparison is computationally expensive, doing aggregation will significantly increase the end-end signal latency. (b) complexity of filter operation — if filter aggregation was realized, the update of one filter request would potentially result in many updates cascading upstream. For example, the single removal of a previous subscription request might break an existing filter aggregation into many individual filters, resulting in a withdraw and several sets of requests going upstream. (c) security considerations — if all filter requests are signed by the end host, FMs at intermediate routers must be able to authenticate and verify the integrity of each request. If AFSPv2 were to aggregate filters, then each active router would need to be able to signal aggregated filter requests, and would require that other active routers be able to authenticate these requests. Although, we are not in favor of putting filter aggregation as part of signaling protocol, filter aggregation will still be useful in FM in order to reduce the height of the filtering tree, accelerating data forwarding.

AFSPv2 must be able to response to topology changes correctly. However, topology changes are considered as rare events. Thus, AFSPv2 must not handle topology change by introducing significant complexity to “normal” signaling process.

ATDP already builds up reliable transmission channels between each active peers. AFSPv2 utilizes these channels for propagating signaling messages.
The organization of this design document is as following. In Section 2, we introduce the inter-
faces between AFSPv2 and the three other modules, and the signaling messages transmitted among
AFSPv2 modules. In Section 3, we describe the active filter signaling protocol version 2 (AFSPv2)
in detail. Finally, we discuss some problems with the current AFSPv2 design in Section 4.

2 Interfaces Between of AFSPv2 and Other Modules

Compared with AFSPv1, AFSPv2 greatly limits the interactions between the AFSP module and the
FM module, and consequently simplifies the interfaces between them. The functionality of AFSPv2
is thus reduced to merely passing filter requests up/down stream. Ideally AFSPv2 should work even
without FM support (unless there is a topology change). Also, by integrating and utilizing ATDP
module, AFSPv2 is alleviated from dynamically tracking underlying multicast topology and reliablly
signaling between (logically) adjacent active nodes. A major focus of this document is to clarify the
logical interfaces between AFSPv2 and other modules.

Before looking into the details of the interfaces between AFSPv2 and other modules, we intro-
duce some terminology that will be used in this document. We refer to the end-hosts running the
four active components, FM, ATDP, AFSPv2 and Application module, and the intermediate active
routers running the active functions of FM, ATDP and AFSPv2 as active nodes. Note that, only
end-hosts contain the application module. Furthermore, the terms parent and child are used to refer
to the nearest upstream or downstream active node on a source-based multicast tree, respectively. A
parent/child need not be the adjacent upstream/downstream nodes in the underlying multicast tree,
since there might be intermediate non-active routers. In addition, the term peer is used to indicate a
neighboring active node, which could be a parent or a child.

In the multicast tree shown in Figure 2, nodes A, B, C and D are end-hosts, and node X is an
intermediate active router. The source-based (A is the source) multicast data dissemination pathes are
shown as directed links in this figure. The end-hosts A, B, C, D and the active router X shown in
Figure 2 are all active nodes. The small black squares in Figure 2 represent the network interfaces
through which the multicast data are disseminated, which can be either the physical interfaces or the
logical interfaces as used in multicast tunneling. In general, we use virtual interface (VIF) to refer to
these network interfaces. For details of virtual interface, please see [1, 3]. In the meantime, AFSPv2s
sitting at end-hosts treat each application as a local VIF (not shown in the figure).
2.1 Information maintained in ATDP and FM

ATDP [1] maintains a mesh of reliable signalling channels between pairs of adjacent AFSPv2 peers, which AFSPv2 uses to exchange signaling messages with its peers. In this respect, AFSPv2 and ATDP interact similarly to the application layer and UDP transport layer in a conventional Internet protocol stack.\(^1\) AFSPv2 provides a message payload and routing information to ATDP, which transmits the payload along with a header to the appropriate peers. Note that a single message offered to ATDP by AFSPv2 may result in multiple transmissions to multiple peers. In the topology of Fig. 2, for example, a natural action for AFSPv2 at node \(X\) might be to send a message to any children along all VIFs excluding VIF \(t\). In this case, ATDP would transmit one copy of the message to each of \(B, C,\) and \(D\). Less obvious is the fact that a single message delivered to an AFSPv2 peer by ATDP might apply to multiple logical peers. A logical peering relationship is a tuple, (source, group, parent, parent_vif, child, child_vif). It is clear that two peers, \(X\) and \(C\), for example, can participate in multiple logical peering relationships. Thus, if \(A\) and \(B\) were both sources, the single message delivered to \(C\) in the preceding example would ‘demultiplex’ into two messages, one for each logical peering relationship. Since ATDP cannot know whether AFSPv2 prefers to treat these two messages logical messages individually or as a single message, it can only provide sufficient header information when a message is received to allow AFSPv2 to perform this demultiplexing itself.

Since ATDP maintains a table of relationships to its neighboring active nodes, it can provide AFSPv2 with notifications about changes of the underlying multicast topology. We will present these notifications in Section 2.2.2 and what AFSPv2 does on receiving them in Section 3.5.

Filter manager (FM) is responsible for setting up data filters with the data forwarding path based on the subscribe requests signaled by AFSPv2. Since AFSPv2 only supports signaling subscription requests from receivers, there is no publication announcement maintained by FM to set up filters. In addition to executing the filter algorithm [2], FM maintains a database containing the following information\(^2\).

\[
\text{filter}_\text{uid}, \text{seq}_\text{no}, \text{filter}_\text{specification}, \text{session}, \text{local}_\text{vif}, \text{up/down}, \text{peer}
\]

where \(\text{filter}_\text{uid}\) is a unique identifier associated with a filter request (e.g., the IP address of the end host + the process ID of the application + a unique ID assigned for the specific filter). This identifier is assigned by the end host application and used to distinguish different filters. The \(\text{seq}_\text{no}\) that is maintained at each end-host-AFSPv2 is incremented monotonically and attached to each filter request from local applications. The \(\text{filter}_\text{specification}\) describes the content attribute associated with a filter. We use \(\text{session}\) to refer to the information describing a multicast group (e.g., group_ip). The \(\text{local}_\text{vif}\) refers to the VIF that the filter is associated with. In addition, the end host treats each application as a special \(\text{local}_\text{vif}\). The \text{up/down} flag indicates whether the filter specification is from downstream (a filter request) or from upstream (an announcement or second pass response of filters potentially installed at the parent peer). At last, \(\text{peer}\) indicates the \(\text{IP}\) address of a neighboring active node from which the corresponding filter information is received. (Note, when \(\text{local}_\text{vif}\) indicates that a filter is from a local application, \(\text{peer}\) could be used to record its port information.)

---

\(^1\)Indeed, a socket-like interface would be a reasonable implementation choice for AFSPv2/ATDP interaction.

\(^2\)Since there is no resource or reference that describes the database of FM, here, what we present are the information that we think should be maintained by FM.
2.2 Interface Between AFSPv2 and Other Modules

2.2.1 Application Module and AFSPv2

In a publish/subscribe system built on an active multicast topology, a subscriber (or a publisher) is an application sitting at an end-host. Thus, the application module generates subscription (filter) requests. A filter request is submitted through a *Subscribe* procedure provided by AFSPv2. This procedure is defined below.

```
Subscribe(session, < filter_uid, filter_specification >)
```

Here, for an existing `filter_uid`, a NULL `filter_specification` indicates that the corresponding filter should be removed; otherwise, the corresponding filter should be updated. On the other hand, if the `filter_uid` is a new one, the corresponding filter should be added.

For simplification, in the rest of this document, we let `filter req` denote the “black box” of the tuple `< filter_uid, filter_specification >`. Note that, at no circumstances, AFSPv2 needs to unpack or modify the tuple.

2.2.2 ATDP and AFSPv2

As discussed in the previous subsection, ATDP maintains the routing table at each active node, and monitors the multicast topology changes. When a topology change is detected, ATDP will pass one of the following messages to AFSPv2 as appropriate:

```
ParentChanged(src, grp, peer, local_vif)
ForefatherChanged(src, grp, peer, local_vif)
ChildAttach(src, grp, peer, local_vif)
ChildDetach(src, grp, peer, local_vif)
```

**Arguments:**

- `src` Sender in a source-based multicast tree.
- `grp` Multicast group address.
- `peer` Contact IP for the parent or child (depending on the notification).
- `local_vif` For `ParentChanged` or `ForefatherChanged`, the VIF on which data forwarded by `peer` arrives. For `ChildAttach` or `ChildDetach`, the VIF on which data destined for `peer` is forwarded.

To send a signaling message, AFSPv2 makes use the following ATDP procedure:

```
atdpSendVif(src, grp, peer, local_vif, direction, mode, msg)
```

**Arguments:**
src  Source of a multicast tree. May be a wildcard (SRC\_ANY).

ggrp  Multicast group address.

peer  Contact IP of receiver or a wildcard (PEER\_ANY).

local\_vif  Identifies a specific local VIF. May be wildcard (VIF\_ANY).

direction  Selects only parents (DIR\_PARENT), only children (DIR\_CHILD), or all peers (DIR\_ANY).

mode  MODE\_DEFAULT explicitly selects peers associated with local\_vif. MODE\_EXCLUDE explicitly excludes peers associated with local\_vif.

msg  The message payload.

To receive a message, AFSPv2 calls the atdpRcv procedure, which provides not only the message itself, but also the values of the ATDP header fields, which can be used to ‘demultiplex’ messages for multiple logical peers.

atdpRcv(&src, &ggrp, &peer, &direction, &remote\_vif, &local\_vif, &msg, &length)

Arguments:

&src  Source of a multicast tree. May be a wildcard (SRC\_ANY).

&ggrp  Multicast group address.

&peer  Contact IP of receiver, never a wildcard.

&direction  Contains the value of &direction argument used by the sender.

&local\_vif  Identifies a specific local VIF. Never a wildcard.

&remote\_vif  Identifies a specific local VIF. Never a wildcard.

&msg  The message payload.

&length  Initially holds the maximum size that can be stored in &msg. On return, holds the actual size stored.

2.2.3  FM and AFSPv2

Filter manager (FM) is the module where active filters are installed and maintained based on the filter requests passed by AFSPv2.

When AFSPv2 gets a filter request, it passes a Set Filter signal to FM, with the following format,

\( \text{Set Filter}(\text{session}, \text{filters}, \text{local}\_vif, \text{up}/\text{down}, \text{peer}) \)
Here, filters is a variable-length list containing tuples of \(<filter\.req, seq\.no >\). When SetFilter is used, FM should record the corresponding filter information in the database and synchronously or asynchronously modify the filter settings.

Due to topology change, AFSPv2 passes a DropFilter message to FM, with the following format,

\[
\text{DropFilter}(\text{session, filters, local}_vif, \text{up/down, peer}).
\]

When DropFilter is used, FM removes the matching filters with the same session, filter\_uuid, local\_vif, up/down flag, and peer information, but having a smaller or equal seq\_no.

In addition, a function call, GetPeerFilter, is designed to allow AFSPv2 obtain filter information maintained in FM. This function call is as follows,

\[
\text{GetPeerFilter}(\text{session, include/exclude, local}_vif, \text{up/down, peer}).
\]

The GetPeerFilter function returns a filter list containing tuples of \(<filter\.req, seq\.no >\). In the arguments, flag include/exclude indicates whether the requested filters should be associated with the local\_vif (include) or the interfaces other than the local\_vif (exclude). Meanwhile, session, up/down flag and peer are the information to be matched. The peer argument can be of three values: a specific peer IP, a wildcard PEER\_ANY, or PEER\_NONLOCAL. When PEER\_NONLOCAL is used, filter requests from local application should be excluded.

### 2.3 Signaling Messages Between Peer AFSPv2s

Other than the signaling messages sent among AFSPv2 and the three other modules, to accomplish filter setup and removal, peer AFSPv2s should also communicate with each other by signaling messages. In our design, we use a two-pass signaling scheme. In the “first pass”, the filter setup requests are sent from the downstream AFSPv2 to its upstream peer AFSPv2. In the “second pass”, an upstream AFSPv2 notifies a downstream AFSPv2 what filters have been requested above. The “second pass” is designed for optimization purpose, given that the knowledge of what data are coming from upstream helps in building a better filter decision tree [2]. Since we do not assume all the multicast enabled routers are active, it is very likely that several downstream nodes connect to the same local\_vif (also true for broadcast media e.g. ethernet). Thus a downstream node can receive and only receive the information requested by itself and its siblings. The “second pass” is used to provide such information. For details, please see [3]. In this document, we call the message of the first pass signaling as \(Q - request\), and the message of the second pass signaling as \(A - message\).

In the example shown in Figure 2, suppose that \(D\) sends a \(Q - request\) to \(X\), indicating to add a filter. On receiving this \(Q - request\), \(X\) sends an \(A - message\) to both \(D\) and \(C\), including not only the filters required by \(D\) but also the filters required by \(C\). This is because that \(C\) and \(D\) connect to the same VIF at \(X\), and data transmitted through VIF \(v\) can arrive at both \(C\) and \(D\).

The messages exchanged between peer AFSPv2s are,

\[
Q(\text{session, filters, ttl})
\]

\[
A(\text{session, filters, update\_method})
\]

\[
D(\text{session, filters, ttl})
\]
Here, the $Q$-request is used to propagate filter requests upstream to data sources; the $A$-message implements the one-hop reflection of the second pass; the $D$ message is designed to withdraw stale filter requests associated with old topology from upstream active routers. In these messages, a $ttl$ field is maintained to avoid looped signaling. The $update\_method$ field can take values of $UPDATE\_INCREMENTAL$ or $UPDATE\_FULL$, where when $UPDATE\_INCREMENTAL$ is used, $A$-message only contains the incremental changes of the filter list, otherwise, $A$-message contains the full filter list. Every time, when AFSPv2 generates one of the above signaling messages, AFSPv2 passes it to ATDP module. ATDP then attaches corresponding routing information (e.g., peer, local_vif, remote_vif, direction etc), to the header of the signaling message, and transmits it through the signaling channel. We will see how these three messages are used to conduct the signaling procedure among AFSPv2s in the next section.

3 AFSPv2 protocol outlines

In this section, we introduce the AFSPv2 signaling protocol in detail. For each AFSPv2 at the active nodes in a multicast topology, the following steps are followed on receiving different messages.

3.1 On receiving a Subscribe call from application:

$Subscribe(session, filter\_req)$

AFSPv2 does the follows:

- increment seq_no
- set filter locally (for the corresponding application)
- forward this request to all parent peers on all interfaces with maximum TTL value

In pseudo code:

```plaintext
++seq_no;
filters=<filter_req, seq_no>); //construct filter list
SetFilter(session, filters, APP, down, APP_port);
atdpSendVif(SRC_ANY, session.g, PEER_ANY, VIF_ANY, DIR_PARENT, 
            MODE_DEFAULT, Q(session, filters, ttl_MAX));
```
3.2 On receiving a $Q$ — request from a peer AFSPv2

$$Q(session, filters, ttl)$$

The following header information is provided by atdpRcv.

$src, grp, peer, local_vif, remote_vif, direction$

AFSPv2 does

- save subscription request and update filter through FM
- decrement the ttl and forward this request to all parent peers on interfaces other than the arrival interface (if ttl is still greater than 0)
- send one-hop reflection — $A$ — messages to all child peers located on the arrival interface

Note that $Q$ — requests are sent through ATDP channels. The interface through which the message is received may be different from the interface through which multicast data toward the child peer are sent. Thus, the arrival interface should be the destination vif specified in the $Q$ — request.

In pseudo code:

/* Q requests are always sent from child to parent, so any other direction is invalid.*/
if (direction != PARENT) return ERROR;

/*ATDP multiplexing sends one message for each unique (local_vif, remote_vif) pair. Therefore, the local_vif will never contain a wildcard and we may take its value to be the arrival vif.*/
SetFilter(session, filters, local_vif, down, peer);

ttl_next = ttl-1;
if (ttl_next > 0)
    /*Send the Q to all parents excluding ones along arrival vifs*/
    atdpSendVif(src, grp, peer, local_vif, DIR_PARENT, MODE_EXCLUDE, Q(session, filters, ttl_next));

    /*Send the A-message with incremental update to all children along arrival vif.*/
    atdpSendVif(src, grp, peer, local_vif, DIR_CHILD, MODE_DEFAULT, A(session, filters, UPDATE_INCREMENTAL));
end if;
3.3 On receiving a D—message from a peer AFSPv2

\[ D(session, filters, ttl) \]

The following header information is provided by atdpRcv.

\[ src, grp, peer, local_vif, peer_vif, direction \]

AFSPv2 does

- drop previous subscription request of the same uuid with equal or smaller sequence number through FM
- decrement the ttl and forward this message to all parent peers on interfaces other than the arrival interface (if ttl is still greater than 0)
- retrieve updated filter settings on the arrival interface and send A—messages to all child peers located on the interface

In pseudo code:

/*D messages are always sent from child to parent so any other direction is invalid.*/
if (direction != PARENT) return ERROR;

/*ATDP multiplexing sends one message for each unique (local_vif, remote_vif) pair. Therefore, the local_vif will never contain a wildcard and we may take its value to be the arrival vif.*/
DropFilter(session, filters, local_vif, down, peer);

ttl_next = ttl-1;
if (ttl_next > 0)
    /*Send the Q to all parents excluding ones along arrival vifs*/
    atdpSendVif(src, grp, peer, local_vif, DIR_PARENT, MODE_EXCLUDE, D(session, filters, ttl_next));

    /*Send an A with full update of filter state to all children along arrival vif.*/
    filters = GetPeerFilter(session, include, local_vif, down, PEER_ANY);
    atdpSendVif(src, grp, peer, local_vif, DIR_CHILD, MODE_DEFAULT, A(session, filters, UPDATE_FULL));
end if;
3.4 On receiving an \textit{A} — \textit{message} from a peer AFSPv2

\[ A(session, filters, update\_method) \]

The following header information is provided by atdpRcv.

\[ src, grp, peer, local\_vif, peer\_vif, direction \]

AFSPv2 does

• store the information (filter installed at parent) contained the \textit{A} — \textit{message} at FM

In pseudo code:

/*A messages are always sent from parent to child, so any other direction is invalid.*/
if (direction != CHILD) return ERROR;
if (update\_method == UPDATE\_INCREMENTAL)
    SetFilter(session, filters, local\_vif, up, peer);
else if (update\_method == UPDATE\_FULL)
    old\_filters = GetPeerFilter(session, include, local\_vif, up, peer);
    DropFilter(session, old\_filters, local\_vif, up, peer);
    SetFilter(session, filters, local\_vif, up, peer);
3.5 On detecting topology changes from ATDP

\[ Parent\text{Changed}(src, grp, peer, local\_vif) \]

- gather subscription requests from all child peers with respect to the new parent, i.e., the child peers connecting to a different interface than the one \(local\_vif\) to which the new parent is attached
- construct \(Q\) — requests and sent to this new parent with maximum ttl value
- increment sequence number
- resend all subscriptions originated from local applications to all parent peers on all interfaces with new sequence number

In pseudo-code

```c
filters = GetPeerFilter(session, exclude, local_vif, down, PEER_NONLOCAL);
atdpSendVif(src, grp, peer, local_vif, DIR_PARENT, MODE_DEFAULT,
            Q(session, filters, ttl_MAX));
seq_no = seq_no + 1;
filters = GetPeerFilter(session, include, APP, down, PEER_ANY);
/* modify filter list with new sequence numbers */
for each tuple <filter\_req, old_seq_no> in filters
    old_seq_no = seq_no;
atdpSendVif(src, grp, peer, local_vif, DIR_PARENT, MODE_DEFAULT,
            Q(session, filters, ttl_MAX));
```

\[ Forefather\text{Changed}(src, grp, peer, local\_vif) \]

- increment sequence number
- resend all subscriptions originated from local applications to all parent peers on all interfaces with new sequence number

In pseudo-code

```c
seq_no = seq_no + 1;
filters = GetPeerFilter(session, include, APP, down, PEER_ANY);
/* modify filter list with new sequence numbers */
for each tuple <filter\_req, old_seq_no> in filters
```
old_seq_no = seq_no;
atdpSendVif(src, grp, peer, local_vif, DIR_PARENT, MODE_DEFAULT,
Q(session, filters, ttl_MAX));

\textit{ChildAttach}(src, grp, peer, local_vif)

- gather all the filter settings associated with the interface to which the child peer is connected
- construct \( A - messages \) and sent to this new child

In pseudo-code

\begin{verbatim}
filters = GetPeerFilter(session, include, local_vif, down, PEER_ANY);
atdpSendVif(src, grp, peer, local_vif, DIR_CHILD, MODE_DEFAULT,
A(session, filters, UPDATE_FULL));
\end{verbatim}

\textit{ChildDetach}(src, grp, peer, local_vif)

- query FM for all the subscription requests sent from this child peer and ask FM to drop corresponding entries
- construct \( D - messages \) and send to all parent peers on different interfaces with maximum ttl value
- retrieve updated filter settings associated with the interface where the missing child peer was connected, send \( A - messages \) to all child peers on the same interfaces

In pseudo-code

\begin{verbatim}
filters = GetPeerFilter(session, include, local_vif, down, problem_peer);
DropFilter(session, filters);
atdpSendVif(SRC_ANY, grp, PEER_ANY, local_vif, DIR_PARENT, MODE_EXCLUDE,
D(session, filters, ttl_MAX));
filters = GetPeerFilter(session, include, local_vif, down, peer_ANY);
atdpSendVif(SRC_ANY, grp, PEER_ANY, local_vif, DIR_CHILD, MODE_DEFAULT,
A(session, filters, UPDATE_FULL));
\end{verbatim}
4 Some Discussions

4.1 reliable data dissemination

In many circumstances, both reliable data dissemination and interests management are desirable. For example, a user might require all its data to be transmitted reliably to anyone with interest in that data. Conceptually, since “Active Filtering” is implemented in the routing layer while reliability considerations should be built in the transport layer or application layer, there should not be much problem to put them together. However, enforcing reliability usually depends on detecting gaps on messages’ sequence numbers, and interest-based filtering can produce discontinuities in sequence numbers. This suggests that straightforward implementation are not workable. Figure 3 illustrates this problem.

Since the underlying multicast topology can change, solutions of storing state information and implementing hop-by-hop reliability at active routes are not feasible.

4.2 per-source tree

When the underlying multicast topology is based on per-source trees instead of a shared tree, potential problems can occur. While Figure 4 (a) shows an efficiency problem (unnecessary filter requests are transmitted), Figure 4 (b) shows a serious loop problem.

In AFSPv1, the loop problem is avoided by letting FM check the filter request first. Only ”new” requests are forwarded. Currently, a TTL field is used to limit the impact of potential loop situation when per-source tree is assumed.

References

Figure 4: Problems caused by per-source multicast tree
