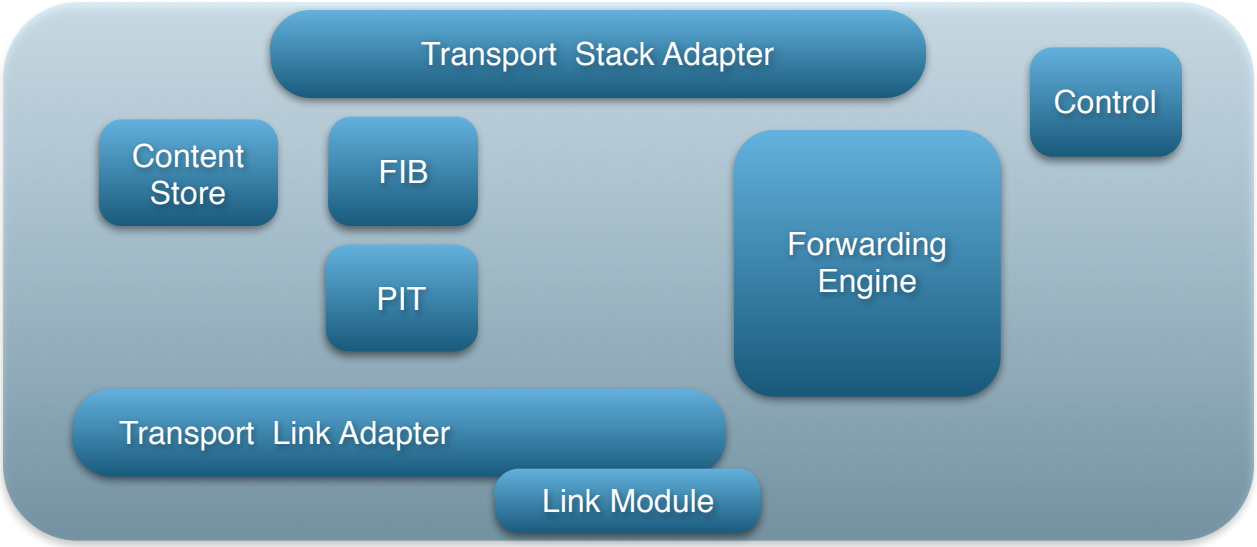
Networking and Distributed Systems
Computing Science Laboratory

CCNx Athena Forwarder

Kevin Fox[1*]

Abstract
The design of Athena as a platform for prototyping CCNx forwarding agents.

[1]*Computing Science Laboratory, PARC*
***Corresponding author**: kevin.fox@parc.com



1. Introduction

Athena is a framework for prototyping and developing CCNx forwarding agents. It's intended to assist research and development teams working on experimental prototyping where protocol correctness is of importance. Although it is not intended as a production deployment vehicle, it may be modeled and tuned to such an extent, its main purpose, however, is to provide the base functionality required to create protocol compliant CCNx forwarders. Modifications and extensions should be easy to implement and able to be assessed with a reasonable amount of flexibility; to that end the framework, implementation and its interdependencies have been kept to a minimum.

Those looking to to develop a performance tuned CCNx forwarder implementation are encouraged to reference the Metis forwarder included in the CCNx 1.0 release.

1.1 Athena Runtime

The Athena runtime provides simple single threaded message processing and forwarding agent. The agent requires a single initial configured link to begin interacting as a CCNx forwarder. This initial link defaults as a tcp listener on localhost port 9695, but can be specifically configured with command line arguments that are processed and instantiated by the Control component. Once an initial link is configured and connected to, further control messages can be sent utilizing that link to expand the configuration of the forwarder (see 2.3).

1.2 **Development Requirements**

Each forwarder component module allows protected concurrent access but has no inherent hard dependency on concurrency. That is, single threaded implementations should be as easily developed as threaded ones.

Each module has been composed with a set of well defined interfaces along with a documented set of requirements and behavior.

Athena requires an initial interface link to enable forwarding operations to begin. It is assumed that any link provided by default, or explicitly created, is secure and intact. Additional links and routing directives may be provided by sending interest control messages after the initial link is established but all links must be explicitly created. There is currently no advertisement or interface probing that is performed by the forwarder.

2. Components

Athena is composed of a main forwarding engine that passes messages, along with their associated link vectors, to a small number of components with well defined interfaces that assist in to determining outgoing message destinations. Applications connect and interact with the forwarder by communicating through a Portal instance which creates a transport stack that attaches to a local Athena instance through a Transport Stack Adapter.

2**.1 TransportStackAdapter**

The Athena Transport Stack Adapter is an external module that coordinates message transmission between the Transport Stack and the Athena forwarder. The Athena forwarder expects all established links to be to an adjacent forwarder and all messages to be exchanged in wire format. The Athena Transport Stack Adapter enables the Transport Stack to look, for all intents and purposes, like another forwarder to Athena, aside from having a hop cost of zero.

The TransportStackAdapter provides message format and content translation if required. Athena currently uses the same Transport Stack Adapter used to support Metis.

2.2 ForwardingEngine

The Forwarding Engine drives the runtime processing of messages through the forwarder. It is aware of CCNxMetaMessages, their types (Interest and Content), their hop-limit and expiration fields and any link vectors associated with the origin or destination of the message. Messages are retrieved by the Forwarding Engine from the TransportLinkAdapter, processed according to CCNx protocol specifications utilizing the PIT, FIB and Content Store components , with the results being dispatched back to the TransportLinkAdapter for delivery through the specified link interfaces.

Messages with a hop-limit of 0 will not be forwarded on links with a non zero hop cost, messages with a past due expiration will be

dropped, and incoming messages that do not pass verification logged and discarded.

2.3 Control

The Control component is responsible for processing and managing control messages that come into, or are generated by, the forwarder. Control messages which involve coordination across components in the forwarder are managed by the Control module, while control messages intended for specific components are passed on to those components to allow them an opportunity to act. Control messages directed at a component that are not not operated on by that component may be handled with default actions by the Control module.

Control messages are delivered as CCNxInterest messages that are addressed to forwarder components and can include extended arguments in their payload. Interest control messages are handled through athenaInterestControl().

```
int athenaInterestControl(Athena *athena,
    CCNxMetaMessage *message,
    PARCBitVector *ingressVector)
```

Messages are initially passed to and allowed to be handled by each forwarder components *_ProcessMessage() method. Messages which are not processed by a component module may be processed by default message handlers in the control module.

lci:/local/forwarder/FIB/add
lci:/local/forwarder/FIB/remove
lci:/local/forwarder/FIB/lookup
lci:/local/forwarder/TransportLinkAdapter/add
lci:/local/forwarder/TransportLinkAdapter/remove
lci:/local/forwarder/TransportLinkAdapter/list
lci:/local/forwarder/ContentStore/resize
lci:/local/forwarder/Control/set
…

2.4 PIT

The PIT maintains pending interest requests for the ForwardingEngine.

```
AthenaPIT *AthenaPIT_Create()

void AthenaPIT_Release(AthenaPIT **)

PARCBitVector *athenaPIT_Match(AthenaPIT
*athenaPIT,
    CCNxContentObject *contentObject,
    PARCBitVector *ingressVector)

AthenaPITResolution
AthenaPIT_AddInterest(AthenaPIT *athenaPIT,
    CCNxInterest *interest,
    PARCBitVector  *ingressVector,
    PARCBitVector, **expectedReturnVector)

bool athenaPIT_RemoveLink(AthenaPIT *athenaPIT,
    PARCBitVector *);

CCNxMetaMessage *athenaPIT_ProcessMessage(
     AthenaPIT *athenaPIT,
     CCNxMetaMessage *interest)
```

athenaPIT_AddInterest creates a new PIT entry, or aggregates the request if one already exists, returning its resolution to forward, not forward or if an error occurred to the forwarding engine .

Each PIT entry contains a link vector indicating which interface link an interest came from. That link information is utilized later to direct where returning Content should be forwarded back to. The PIT also maintains an expected return vector which documents the links that interests were originally forwarded to. The PIT uses the expected return vector to verify that the link returning the Content is one of the same links that the interest was originally forwarded out on. If it finds that the content arrived from a link interface not associated with an outgoing interest, the occurrence is logged and the Content dropped. The PIT also maintains an expiration time for each PIT entry and ensures that expired PIT entries are discarded.

athenaPIT_Match returns the link vector to use to route content objects. I will return a NULL vector if there is no PIT entry, the PIT entry has expired or the ingress vector did not match the the expectedReturnVector provided to **athenaPIT_AddInterest** when the original interest was sent out.

When an link is removed by the forwarder, either explicitly or because it was closed or an error occurred, **athenaPIT_RemoveLink** is called on the PIT to request that the link(s) specified in the vector be removed and scrubbed from all PIT entries. New PIT entries will not be created with the specified links until this call successfully returns.

2.5 FIB

The FIB is responsible for maintaining a forwarding table that maps CCNxMetaMessages to their corresponding egress link vector. Routing entries are added and removed by sending interest control messages.

```
AthenaFIB *athenaFIB_Create()

void athenaFIB_Release(AthenaFIB **)

PARCBitVector *athenaFIB_Lookup(
    AthenaFIB *athenaFIB,
    CCNxName *ccnxName)

bool athenaFIB_AddRoute(
    AthenaFIB *athenaFIB,
    CCNxName *ccnxName,
    PARCBitVector *egressVector)

bool athenaFIB_DeleteRoute(
    AthenaFIB *athenaFIB,
    CCNxName *ccnxName,
    PARCBitVector *egressVector)

bool athenaFIB_RemoveLink(
    AthenaFIB *athenaFIB,
    PARCBitVector *egressVector)

CCNxMetaMessage *athenaFIB_ProcessMessage(
    AthenaFIB *athenaFIB,
    CCNxMetaMessage *message)
```

New routes are added to a FIB using **athenaFIB_AddRoute** specifying the CCNxName to match and a vector of one or more links that any matching messages should be forwarded to.

athenaFIB_DeleteRoute removes the specified links on an existing route entry, and may remove the route entirely if no links remain in a routing entry.

athenaFIB_Lookup returns the current egress link vector for the FIB entry for a matching message if one is found.

When an link is removed by the forwarder, **athenaFIB_RemoveLink** is called on the FIB to request that the link(s) specified in the vector be removed and scrubbed from all FIB entries. New FIB entries will not be created with the specified links until this call successfully returns. Routes which no longer contain any egress links are deleted.

2.6 ContentStore

The Content Store is an optional component primarily responsible for caching Content Object messages for the forwarder.

```
AthenaContentStore *athenaContentStore_Create(
    AthenaContentStoreInterface *interface,
    AthenaContentStoreConfig *config)

void athenaContentStore_Release(
    AthenaContentStore **store)

bool athenaContentStore_SetCapacity(
    AthenaContentStore *store, size_t maxSizeInMB)

bool athenaContentStore_PutContentObject(
    AthenaContentStore *store,
    CCNxContentObject *contentObject)

CCNxContentObject *athenaContentStore_GetMatch(
    AthenaContentStore *store,
    CCNxInterest *interest)

CCNxMetaMessage
*athenaContentStore_ProcessMessage(
    AthenaFIB *athenaFIB,
    CCNxMetaMessage *message)
```

athenaContentStore_Create is passed the maximum capacity, in mega-bytes, that it is allowed to allocate for its cache. **athenaContentStore_SetCapacity** can be used to dynamically change the amount of memory used by the content store.

There are no guarantees that messages which are given to the Content Store are stored or, even if they are, retrievable.

The Content Store receives **athenaContentStore_GetMatch** calls from the forwarder with interest messages in expectation of retrieving a content object message which was previously stored as the result of a **athenaContentStore_PutContentObject** call.

The Content Store should never return items which have expired, nor does it have any responsibility for storing expired content which has been submitted.

2.7 TransportLinkAdapter

The TransportLinkAdapter is a layered interface between the Forwarding Engine and link specific modules which perform the physical message transmission.

New links are created by sending interest control messages with link specific arguments addressed to link specific modules. For example, the following is addressed to the TCP module and intended to create a localhost listener on port 9696 that will accept tcp tunnel connection requests:

tcp://localhost:9696/listener

The following creates a tunnel link connection to a listener on port 9696:

tcp://localhost:9696

Creating both of these links within the same forwarder creates a loopback connection:

athena -c tcp://localhost:9696/listener -c tcp://localhost:9696/name=Loopback

The following URI creates a tunnel connection to parc.com at port 9695 that can be reference locally by the link name "PARC" and appears to the forwarder as a local link.

tcp://parc.com:9695/name=PARC/local=true

Interest control messages can only reference link instances by name. A name, if provided, must be unique within the context of the Athena instance otherwise it will fail. A default unique name is created by the TransportLinkModule if none is specified.

The TransportLinkAdapter parses the link request schema and passes them onto their associated TransportLinkModule. Arguments following the URL schema are interpreted only by the link specific TransportLinkModule. When a new link is established by a TransportLinkModule, it passes a reference to its newly constructed TransportLink instance back to the TransportLinkAdapter. The TransportLinkAdapter uses the provided TransportLink instances access methods and state to send and receive messages on the link as well as to manage it.

When the TransportLinkAdapter receives a new TransportLink, it inserts the link into its instance list and sets its state to pending. It then sends a message to the Control module with information about the new link and awaits a confirming message before it moves the link from pending to open. Once a link is open, messages may begin being sent and received over it.

The current Athena implementation assumes all link requests are valid and have been authenticated, so no authorizing control messages are currently exchanged.

Links are referenced internally by an index identifier which correlates with the TransportLinkAdapters instance list. This link ID is maintained in a parcBitVector that is passed between Athena components to populate routing table entires, pending interest source and return vectors, with the associated physical links that the TransportLinkAdapter manages. Link IDs can change or be dynamically removed and are only used internally to the forwarder. Link IDs should never be referenced outside of their associated Athena instance, links are referenced externally by referring to them by the unique name they were created with.

When a link is explicitly closed because of a control message or because of an error detected by a Link Module, the TransportLinkAdapter sets its state to pending and calls the Link Modules close routine to allow it to shutdown completely. It then sends a control message to the Control module asking that the link be removed and awaits a confirming message that all references to the link have been scrubbed before removing the link instance. Once a link instance has been removed, the link ID can be reused for another instance. This keeps the instance list and associated Bit Vectors only as large as is necessary.

```
AthenaTransportLinkAdapter *
athenaTransportLinkAdapter_Create(
  AthenaTransportLinkAdapter_RemoveLinkCallback
*removeLinkCallback)

void
athenaTransportLinkAdapter_Destroy(
  AthenaTransportLinkAdapter **)

int
athenaTransportLinkAdapter_Open(
  AthenaTransportLinkAdapter *athenaTransportLinkAdapter,
  PARCURI *connectionURI)

void
athenaTransportLinkAdapter_Poll(
  AthenaTransportLinkAdapter *athenaTransportLinkAdapter,
  int timeout)

PARCBitVector *
athenaTransportLinkAdapter_Close(
  AthenaTransportLinkAdapter *athenaTransportLinkAdapter,
  PARCBitVector *linkVector)

CCNxMetaMessage *
AthenaTransportLinkAdapter_Receive(
  AthenaTransportLinkAdapter *athenaTransportLinkAdapter,
  PARCBitVector **ingressVector,
  int timeout)

PARCBitVector *
athenaTransportLinkAdapter_Send(
  AthenaTransportLinkAdapter *athenaTransportLinkAdapter,
  CCNxMetaMessage *ccnxMessage,
  PARCBitVector *egressLinkVector)

int
athenaTransportLinkAdapter_CloseByName(
  AthenaTransportLinkAdapter *athenaTransportLinkAdapter,
  const char *linkName)

const char *
athenaTransportLinkAdapter_LinkIdToName(
  AthenaTransportLinkAdapter *athenaTransportLinkAdapter,
  unsigned linkId)

unsigned
athenaTransportLinkAdapter_LinkNameToId(
  AthenaTransportLinkAdapter *athenaTransportLinkAdapter,
  const char *linkName)

static int
_athenaTransportLinkAdapter_AddLink(
  AthenaTransportLinkAdapter *athenaTransportLinkAdapter,
  AthenaTransportLink *newTransportLink)
```

[details in the TransportLinkAdapter design document]

2.8 External Dependencies

Athena depends on modules and methods provided by Libparc and Libccnx.

Messages are passed internally as CCNxMetaMessages, with their data fields referenced using the following Libccnx routines.

```
bool
ccnxMetaMessage_Validate(
    CCNxMetaMessage *ccnxMetaMessage)

uint32_t
ccnxInterest_GetHopLimit(
    CCNxInterest *ccnxInterest)

void
ccnxInterest_SetHopLimit(
    CCNxInterest *ccnxInterest,
    uint32_t hopLimit)


uint64_t
ccnxContentObject_GetExpiryTime(
    CCNxContentObject *ccnxContentObject)


void
ccnxContentObject_SetExpiryTime(
    CCNxContentObject *ccnxContentObject, uint64_t time)

CCNxMetaMessageType *
ccnxMetaMessage_GetType(
    CCNxMetaMessage *ccnxMetaMessage)

CCNxName *
ccnxInterest_GetName(
    CCNxInterest *ccnxInterest)

CCNxName *
ccnxContentObject_GetName(
    CCNxContentObject *ccnxContentObject)
```

Interest Control message payloads are used to pass link configuration requests to link specific modules,. Likewise, replies to configuration requests are returned in the payload contents of the ContentObject response. These messages are constructed and payloads retrieved using the following Libccnx interfaces:

```
CCNxInterest *
ccnxInterest_CreateSimple(CCNxName *name)

CCNxInterest *
ccnxInterest_SetPayload(
  CCNxInterest *interest, PARCBuffer *payload)

CCNxContentObject *
ccnxContentObject_CreateWithDataPayload(
  CCNxName *name, PARCBuffer *payload)

PARCBuffer *
ccnxContentObject_GetPayload(
  CCNxContentObject *contentObject);

PARCBuffer *
ccnxInterest_GetPayload(
  CCNxInterest *interest);
```

CCNxMetaMessages are converted to and from wire format inside link specific modules in the TransportLinkAdapter using the following:

```
CCNxMetaMessage *
ccnxMetaMessage_CreateFromWireFormatBuffer(
    PARCBuffer *rawMessage)

PARCBuffer *
ccnxMetaMessage_CreateWireFormatBuffer(
    CCNxMetaMessage *message,
    PARCSigner *signer)
```

Links are referenced internally between forwarder components using PARCBitVector instances:

```
PARCBitVector *parcBitVector_Create()

void parcBitVector_Release(PARCBitVector **)

PARCBitVector *parcBitVector_Set(
    PARCBitVector *parcBitVector, unsigned bit)

unsigned parcBitVector_Get(
    PARCBitVector *parcBitVector, unsigned bit)

PARCBitVector *parcBitVector_SetVector(
    PARCBitVector *parcBitVector,
    PARCBitVector *setBits)

PARCBitVector *parcBitVector_Clear(
    PARCBitVector *parcBitVector, unsigned bit)

PARCBitVector *parcBitVector_ClearVector(
    PARCBitVector *parcBitVector,
    PARCBitVector *clearBits)

unsigned parcBitVector_NumberOfBitsSet(
    PARCBitVector *parcBitVector)

unsigned parcBitVector_NextBitSet(
    PARCBitVector *parcBitVector
    unsigned index)

char *parcBitVector_ToString(
    PARCBitVector *parcBitVector)
```

PARCBitVector is a dynamically sized bit vector container. Individual bits can be set and cleared, or vectors can be used to specify sets of bits to be set or clear. **parcBitVector_NextBitSet** returns the index of the next bit which is set at, or after, the specified index. The number of currently set bits returned by **parcBitVector_NumberOfBitsSet** can be used to bound the number of calls to **parcBitVector_NextBitSet** that are required to retrieve the entire list of bits set in the vector.