The Backbone-Cluster Protocol

Wittawat Tantisiriroj, Jorg Liebeherr, MNG Group
Thursday, February 07, 2008

Table of Contents

1. Introduction

The backbone-cluster (B-CT) protocol realizes a hierarchical peer network topology with a two-layer hierarchy. The lower layer of the topology is built by the Cluster (CT) protocol and the upper layer is built by any other overlay protocol. Recall that the CT protocol organizes sets of nodes of an overlay network in star topologies, called *clusters*. The node in the center of a cluster is called *cluster head* and the nodes at the periphery are called *cluster members*. In the CT protocol, application data is exchanged only between nodes in the same cluster. The B-CT protocol connects multiple clusters and enables the exchange of application data between nodes in different clusters. This is done by connecting the cluster heads by another overlay topology, referred to as the *backbone*.

Figure 1 depicts a network that is created by the B-CT protocol. The figure shows a set of clusters that are each established by the CT protocol. Cluster members and heads are labeled with an `M' and `H', respectively. The head in each cluster is connected to the backbone. The backbone can be constructed using any of the existing overlay protocols, such as the spanning tree protocol, Pastry, or the cluster protocol.

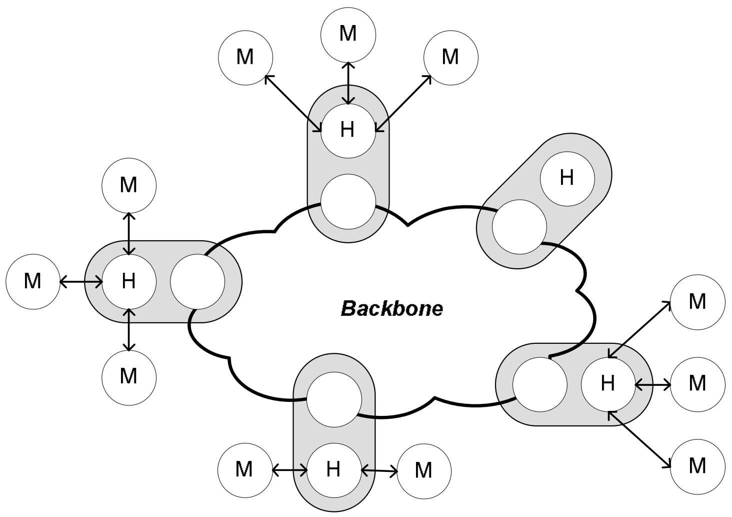

Figure 1. Cluster-Backbone topology.

In the B-CT protocol, each node runs an unmodified instantiation of the CT protocol and the backbone protocol. In this way, the B-CT protocol can add clusters to the periphery of any existing overlay protocol. The B-CT protocol provides the glue between the state machines for the two protocol instantiations.

In a backbone-cluster network, all clusters and the backbone belong to the same overlay network, in the sense that they have the same overlay identifier and compatible configurations. The logical address of a node in a B-CT protocol is the result of concatenating the logical addresses in the backbone and cluster. All nodes in the same cluster have a common prefix which is the backbone address of the cluster head.

The B-CT protocol performs the following tasks:

- It creates the logical address of the node by concatenating the logical address in the backbone with the logical address in the cluster. Cluster members must obtain the backbone prefix from their respective cluster head.
- It builds the neighborhood association of the node from the neighborhood table in the backbone protocol node and the cluster protocol.
- It makes all cluster heads join the backbone topology.
- It disseminates information about cluster heads throughout the backbone by having cluster heads exchange referral lists (see CT protocol) to neighbors in the backbone network.

2. Operations

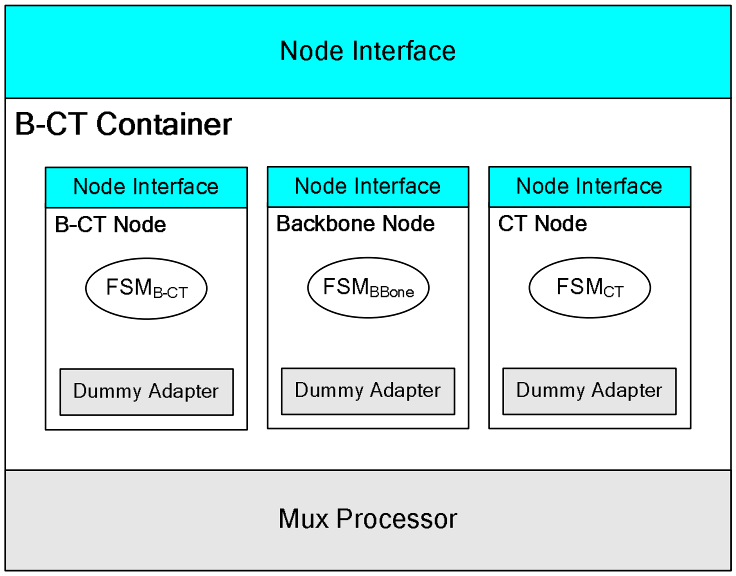

Figure 2. Node in the B-CT Protocol (Cluster head).

The structure of a protocol node in the B-CT protocol is shown in Figure 2. The node contains a node of the backbone protocol and a node of the CT protocol. However, cluster members do not join the backbone overlay topology. In these nodes, the backbone node is inactive.

The B-CT node exploits a property of the overlay node design that permits an overlay node to contain multiple internal overlay nodes. Internal protocol nodes communicate via a virtual adapter, called a *dummy adapter*, and protocol messages from the internal protocol nodes are multiplexed on a single node adapter using an encapsulation header. The encapsulation is performed by an additional protocol, called the Multiplexing (MUX) protocol. The MUX protocol and the dummy adapter are discussed elsewhere. In the design of the B-CT protocol, a B-CT node is a container with one or two internal nodes. Each B-CT node includes a node of the CT protocol and node of the backbone protocol. The CT node is always active. The backbone node is active only, if the CT node is running as a cluster head.

As Figure 2, that a B-CT node may include up to three finite state machines (FSMs). The finite state machines of the cluster node and the backbone nodes, denoted by FSM_{BBone} and FSM_{CT}, execute independent of each other and are unaware of the presence of the B-CT node. A B-CT node runs a finite state machine, FSM_{B-CT}, that takes into consideration the state of the internal nodes. The state machine of the B-CT node, explained in detail in the next section, is minimal. The node periodically tests the state of the cluster node. When the cluster node is running as a member, the B-CT node sends a message to the cluster head to request backbone address of the cluster head. When the cluster node is running as a cluster head, the B-CT protocol creates a backbone node and maintains the membership of the backbone node in the backbone network.

The logical address of an overlay node serves as a unique address in the overlay network. The logical address of B-CT node is constructed as a concatenation of the logical address of a backbone address, LA_{BBone}, and a cluster address, LA_{CT}. This concatenation is denoted as $LA_{B-CT} = LA_{BBone} \circ LA_{CT}$.

The B-CT protocol has only a single exchange of protocol messages, whose purpose is to assign a backbone logical address LA_{BBone} to a cluster member. A cluster member sends a request for a backbone address to its cluster head. The cluster head returns its address in the backbone network.

In HyperCast, application data is forwarded in spanning trees that are embedded in the overlay topology. Each node forwards data to one or more of its neighbors in the overlay topology. A multicast message is forwarded downstream in a rooted tree that has the sender of the multicast message as the root. A unicast message is forwarded upstream in a rooted tree that has the destination of the message as the root. In HyperCast, each node can locally compute its upstream neighbor (parent) and its downstream neighbors (children) with respect to a given root note. In the B-CT protocol, the computation must consider neighbors in a cluster topology as well as in the backbone topology. If a B-CT that contains a CT node that is a cluster member, the situation is simple since a node has only one neighbor, i.e., its cluster head. Let us now look at a B-CT node that contains a cluster head. The neighborhood of this node is the union of the neighbors in the cluster and the backbone network. When this node calculates the downstream neighbors in the overlay network, the distinguishes whether the root node is one of its cluster members. If so, the downstream neighbors are all other cluster members and all downstream neighbors in a spanning tree that has the local node as the root. If the root is not one of its cluster members, then the downstream nodes consists of its downstream neighbors with respect to the given root in the backbone and all cluster members. An upstream neighbor is either one of its cluster members (if the root is in its cluster), the upstream node in the backbone network (if the root node is not one of its cluster members).

3. States and State Transitions

In this section, we specify the finite state machines of the B-CT protocol. Table 1 summarizes the states of the protocol. Each B-CT node instantiates a node of the CT protocol. The states of the B-CT protocol depend on the state of the CT protocol. If the CT node s a cluster member, the B-CT node must acquire a backbone logical address (backbone ID) from its cluster members. In the state, *Member without Backbone ID* a node requests a backbone ID from the cluster head. In this state, without a backbone ID, the node has not a valid logical address that can be used as source or destination in the overlay network. In the B-CT protocol, a node must monitor the state of the CT node. This is enforced through a timer, called the Update timer.

Table 1. State Description.

State Name	**State Definition**
Stopped	The node is not running
Unknown	Its cluster node is neither a cluster head or cluster member
Member without Backbone ID	Its cluster node is a cluster member and it does not have a backbone ID
Member without Backbone ID	Its cluster node is a cluster member and it does have a backbone ID
Head	Its cluster node is a cluster head

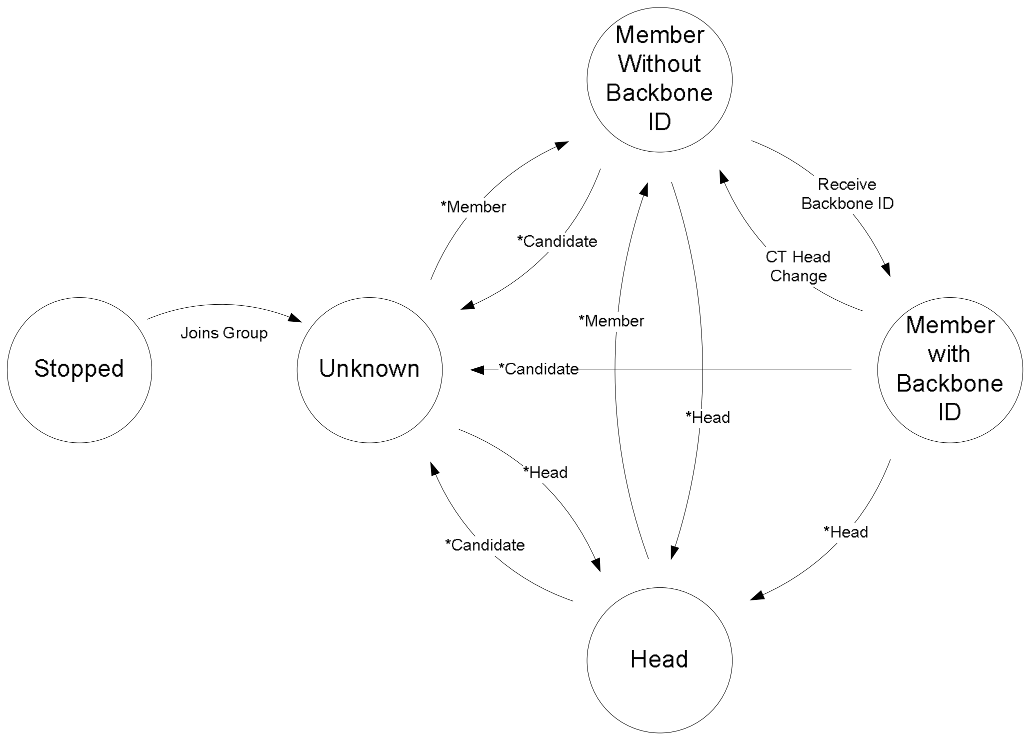

Figure 3. State transition diagram of the Backbone-Cluster.
(From each state, there is an additional edge with label Leave overlay to state Stopped.)

The state transitions are depicted in the diagram in Figure 3. The labels of the links are as follows:

*Candidate=	The state of the CT node changes to *Member Candidate Without Head* or *Member Candidate With Head*
*Member =	The state of the CT node changes to *Member*
*Head =	The state of the CT node changes to *Head Without Member* or *Head With Member*

The following event-action tables provide a detailed description of the B-CT protocol. In the table, CT-State denotes the states of the node in the CT protocol.

State: **Stopped**

Event	*Action*
Join Group	Cluster node performs a join operation → **Unknown**

State: **Unknown**

Event	*Action*
Update timer expires	Check the state of the cluster node If CT-State[1] = **Member** → **Member without Backbone ID** Else if CT-State = **Head Without Member** or **Head With Member** Backbone node joins the backbone topology Update the logical address → **Head**
Leave Group	Leave overlay of CT node and backbone node → **Stopped**

State: **Member without Backbone ID**

Event	*Action*
Update timer expires	Send a *BackboneIDRequest* message to the cluster head ——— Check the current state of the cluster node If CT-State = **Member Candidate Without Head** or **Member Candidate With Head** → **Unknown** Else if CT-State = **Head Without Member** or **Head With Member** Backbone node joins the backbone topology Update the logical address → **Head**
BackboneIDReply message received from cluster head	Update the logical address → **Member**
Leave Group	Leave overlay of CT node and backbone node → **Stopped**

State: **Member with Backbone ID**

Event	*Action*
Update timer expires	If CT-Head Change → **Member without Backbone ID** ——— Check the current state of the cluster node If CT-State = **Member Candidate Without Head** or **/Member Candidate With Head** → **Unknown** Else if CT-State = **Head Without Member** or **Head With Member**

Comment: Currently implemented and still in testing phase.

Comment: Done

[1] CT-State refers to the current state of cluster node in the Tier-Cluster node.

	Backbone node joins the backbone topology Update the logical address → **Head**
Leave Group	Leave overlay of CT node and backbone node → **Stopped**

State: **Head**

Event	**Action**
Update Timer expires	Send a *CT Referral* message containing a list of the backbone neighbors to all cluster nodes. Check the current state of the cluster node If CT-State = **Member Candidate Without Head** or **Member Candidate With Head** Leave the backbone overlay Set a logical address to be undefined → **Unknown** Else if CT-State = **Member** Leave the backbone overlay → **Member without Backbone ID**
BackboneIDRequest message received from a cluster member	Send a *BackboneIDReply* message to the cluster member
Leave Group	Leave overlay of CT node and backbone node → **Stopped**

Comment: This feature is not implemented. We have not decided whether we want to implement it or not.
Issue:
- Which sub-node sends this referral message? B-CT node or CT Node.

For a B-CT node case, how can the B-CT node extract information from CT Node and how the B-CT node inject the information back to CT Node?

For a CT Node case, how can B-CT node force CT Node to send a referral message?

4. Message Formats

This section list the detailed message formats used in the Backbone-Cluster Protocol. The common format for all protocol messages is shown in Figure 4.

1 byte	4 bytes	PASize	PASize	
Type	Overlay Hash	Src PA	Dst PA	Type dependent part

Figure 4. Protocol message of the B-CT protocol.

Comment: Should protocol field and message size be part of message format?
Check with SPT implementation and document.

Comment: What is the Src LA/Dest LA? In the B-CT protocol, the LA consists of backbone+cluster ID. Here, the node may not have a backbone ID.

Oct 31, 2006: delete src and dest La FROM messages. The reason is that source does not have complete LA, and does not know destination LA.

Comment: A sender of a message does not know its Backbone ID and destination's backbone Id. So, Src and Dest LA are undefined. In the current implementation, these field are omitted.

Type: The types of Backbone-Cluster protocol messages are shown in Table 2.

Table 2. Protocol Message Types.

Message Type	Type Field
BackboneIDRequest	0
BackboneIDReply	1

Overlay Hash: A 4-byte long hash value that is derived from all attributes specified in *HashAttributes* of the configuration file.
Src PA: A physical address of the source node.
Dest PA: A physical address of the destination node.

4.1. BackboneIDRequest Message

Comment: Oct 31: JL + WT: delete srs LA and dest LA from all messages. Keep src/dest PA

1 byte	4 bytes	PASize	PASize
0	Overlay Hash	Src PA	Dst PA

Figure 5. BackboneIDRequest message.

4.2. BackboneIDReply Message

1 byte	4 bytes	PASize	PASize	BLASize
1	Overlay Hash	Src PA	Dst PA	Backbone LA

Figure 6. BackboneIDReply Message Format.

Comment: BLASize "The size of the logical address in the backbone network" is removed because all nodes use the same backbone protocol, so it can figure out the BLASize by itself.

Backbone LA: A logical address in the backbone network.

5. Attributes

Example

```
<Node>
    <BC_Container>
        <MainNode>
        <BC>
            <HeartbeatTime>5000</HeartbeatTime>
            <Verification>neighborcheck</Verification>
            <StatName>Node</StatName>
        </BC>
    </MainNode>
    <BackboneNode>
        <DTBroadcast>
            <CacheFile>.Cachefile</CacheFile>
            <TimeoutTime>10000</TimeoutTime>
            <FastHeartbeatTime>250</FastHeartbeatTime>
            <SlowHeartbeatTime>2000</SlowHeartbeatTime>
            <Verification>neighborcheck</Verification>
            <BeaconTime>250</BeaconTime>
            <StatName>Node</StatName>
            <Coords>
                <FIXED>
                    <coordinate>500,500</coordinate>
                </FIXED>
            </Coords>
        </DTBroadcast>
    </BackboneNode>
    <ClusterNode>
        <CT>
            <HeartbeatTime>1000</HeartbeatTime>
            <MemberTimeout>3000</MemberTimeout>
            <HeadTimeout>3000</HeadTimeout>
            <MemberReferralInterval>5000</MemberReferralInterval>
            <HeadCacheReferralInterval>1000</HeadCacheReferralInterval>
            <CacheEntryTimeout>10000</CacheEntryTimeout>
            <OfferCollisionWindow>500</OfferCollisionWindow>
            <Verification>neighborcheck</Verification>
            <StatName>Node</StatName>
            <ReferralEnable>true</ReferralEnable>
            <LimitedReferralSize>1</LimitedReferralSize>
            <CacheFile>.CT_CacheFile</CacheFile>
            <HeadCacheSize>10</HeadCacheSize>
            <HeadNum>1</HeadNum>
            <Head>
                <UnderlayAddress>
                    <INETV4AndOnePort>127.0.0.1:9800</INETV4AndOnePort>
                </UnderlayAddress>
            </Head>
            <Type>
                <MemberOrHead>
                    <Criteria>
                        <Member>
                            <MinimumAvailableMember>1</MinimumAvailableMember>
                            <MaximumMember>20</MaximumMember>
                        </Member>
                        <Location>
                            <Coordinate>-95.2631, 38.9605</Coordinate>
                            <MaxDistance>100</MaxDistance>
```

Comment: The structure changed a bit

```
                        </Location>
                        <Bandwidth>
                            <MinimumRate>56</MinimumRate>
                            <OfferRate>56</OfferRate>
                        </Bandwidth>
                        <Availability>
                            <MinimumValue>5</MinimumValue>
                            <OfferValue>5</OfferValue>
                        </Availability>
                    </Criteria>
                    <SelectionPolicy>
                        <NextFit/>
                    </SelectionPolicy>
                </MemberOrHead>
            </Type>
        </CT>
    </ClusterNode>
  </BC_Container>
</Node>
```

6. Statistics

The Backbone-Cluster protocol supports the following statistics.

Required by M&C

- LogicalAddress (R)
- PhysicalAddress (R)
- NumOfNeighbors (R)
- NeighborTable (R)

Roots of its components statistics

- BackboneNode (R)
- ClusterNode (R)

Time

- NodeStartTime (R)
- NodeStopTime (R)
- HeartbeatTime (RW)

Status

- State (R)
    - Stopped
    - Unknown
    - MemberWithoutBBoneID
    - MemberWithBBoneID
    - Head

Comment: Add detail for each statistic & think about any useful statistics

(R) stands for Read-Only
(RW) stands for Read&Write

Comment: All statistics are implemented.

7. Appendix Implementation of the I_Node and AdapterCall Back Interfaces

`I_LogicalAddress` **`createLogicalAddress`**`(byte[] laddr, int offset)`

Creates a logical address object from a byte array.

```
bla = backbone.createLogicalAddress(laddr, offset)
cla = cluster.createLogicalAddress(laddr, offset+ bla.getSize())
return new TC_LogicalAddress(bla, cla);
```

`I_LogicalAddress` **`createLogicalAddress`**`(java.lang.String laStr)`

Creates a logical address object from a String.

```
String las[] = laStr.split(":");
bla = backbone.createLogicalAddress(las[0]);
cla = cluster.createLogicalAddress(las[1]);
return new TC_LogicalAddress(bla, cla);
```

`I_AddressPair[]` **`getAllNeighbors`**`()`

Returns the node's neighbors' physical/logical address pairs.

Neighbors_B-CT = Neighbors_BBone ∪ Neighbors_CT

`I_AddressPair[]` **`getChildren`**`(I_LogicalAddress root)`

Returns the node's children's physical/logical address pairs, with respect to the spanning tree rooted at `root`.

If node is CT_Head:
Children_B-CT (root) = Children_BBone (FirstElement(root)) ∪ Neighbors_CT

If node is CT_Member:
Children_B-CT (root) = (root == self? Neighbors_CT: nil)

`I_AddressPair` **`getMyAddressPair`**`()`

Returns this logical and physical addresses of this node.

`The following is not correct since it ignores the physical address:`

If node is CT_Head:
Children_B-CT (root) = getMyAddressPair_BBone () ° getMyAddressPair_CT ()

If node is CT_Member:
Children_B-CT (root) = LocalAddress of BBone () ° getMyAddressPair_CT ()

`I_AddressPair[]` **`getParent`**`(I_LogicalAddress root)`

Returns the addresspair of the next hop for a message routed by this node towards the root.

If node is CT_Head:
Parent_B-CT (root) = (root = X and X ∈ Neighbors_CT? X : nil)

If node is CT_Member:
Parent_B-CT (root) = (root == self? Nil : Neighbors_CT)

`boolean` **prevhopCheck**`(I_LogicalAddress src, I_LogicalAddress dst, I_LogicalAddress prehop)`

Check if previous hop is a valid sender.

```
return backbone.prevhopCheck(src, dst, prehop) || cluster.prevhopCheck(src, dst, prehop)
```

`void` **setLogicalAddress**`(I_LogicalAddress la)`

Sets the logical address to specified one.

```
TC_LogicalAddress tcla = (TC_LogicalAddress) la;
backbone.setLogicalAddress(tcla.getBackboneLA());
cluster.setLogicalAddress(tcla.getClusterLA());
```

`void` **setNotificationHandler**`(NotificationHandler nh)`

Set notification handler.

```
this.nh = nh;
backbone.setNotificationHandler(nh);
cluster.setNotificationHandler(nh);
```

`void` **messageArrivedFromAdapter**`(I_Message msg)`

Handles the incoming message (unicast, server, or multicast).

```
If type of msg is BackboneIDRequest and node is CT_Head
    Send a BackboneIDReply
Else type of msg is BackboneIDReply and node is CT_Member
    Update a backboneID.
```

`I_Message` **restoreMessage**`(byte[] receiveBuffer, int[] validBytesStart, int validBytesEnd)`

Creates a message from bytes in a buffer.

```
return TC_Message.restoreMessage(receiveBuffer, validBytesStart, validBytesEnd, m_adapter, config.getOverlayHash());
```

`void` **timerExpired**`(java.lang.Object timerID)`

Handles the arrival of a timer