CSS499 Summer 2006
AgentTeamwork Inter-Cluster Job Resumption

Final Report: Fri 9/15/06

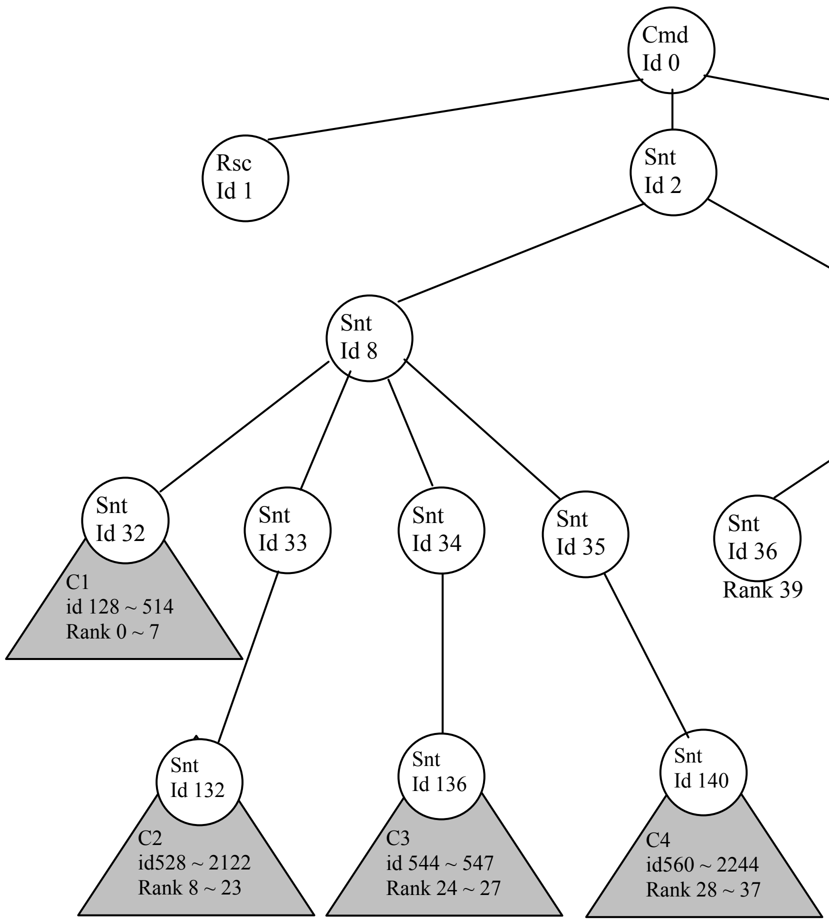
Introduction

In this CSS499 project, two new enhancements were made to the AgentTeamwork:

- The Bookkeeper agent was enabled to work with inter-cluster agents.
- AgentTeamwork was enhanced to support job resumption over multiple clusters.

This report will first give an overview of two key AgentTeamwork features that support inter-cluster deployment and job resumption: the Agent tree structure, and the SentinelAgent args[] structure.
Next it will discuss the major code changes implemented in CommanderAgent.java, SentinelAgent.java, and AgentUtil.java.

Overview of the AgentTeamwork Agent Tree Structure



(Used from Prof. Fukuda's InterClusterDeployment document)

Node #0 is the Commander agent (CommanderAgent.java). It always launches exactly three children: the Resource agent (Node #1), the root Sentinel agent (Node #2), and the root Bookkeeper agent (Node #3). Note that these first four agents must run on public nodes. *See Cuong Ngo's Summer 2006 project report for more details of the Resource Agent.*

Node #2 (the root Sentinel agent) is the parent of all the other local and remote sentinel agents. It does not run a worker process and does not have a regular MPI rank. It does however have a simulated MPI rank of -1 and runs a stub user process (UWAgentGatewayUserApp.java) to manage GridTCP connections and to forward child/parent messages.

It launches up to two children/subtrees. The left subtree (if present) contains all the remote compute clusters. The right subtree (if present) contains the local/public compute nodes.

Node #8 is the gateway sentinel node for the Cluster #1 (C1). It runs on the gateway machine for that cluster. As with the root sentinel agent, it is not a compute node and does not have a regular MPI rank, but has a simulated negative MPI rank (-2 in this case) and runs the stub user process as above.

Its leftmost subtree (headed by the child node #32) contains the compute nodes within that cluster. In this case, there are eight of them. They each run a SentinelAgent that launches a worker process on that node to execute the user's application. Node#32 (which is assigned MPI Rank 0) is the first compute node in the cluster. It has four children in this case (Node ID's 128-131) with MPI ranks 1-4. Node#128 then has three children of its own (Node ID's 512-514) with MPI ranks 7-8.

Now since Cluster#1 in this example only had 8 available compute nodes, and the user actually wanted to use 42 compute nodes, additional remote clusters are needed. These are spawned as children and grandchildren of the first gateway sentinel (Node#8). Each cluster-gateway sentinel's leftmost child/subtree contains the local compute nodes within that cluster, while the rightmost children are the gateway sentinels for additional clusters. Since each node can have up to 4 children, each gateway sentinel can then manage up to 3 additional clusters as direct children of it.

So in this example, **Node #33** is the cluster-gateway sentinel for Cluster#2, and is itself structured similarly to the first cluster. It has 16 compute nodes in its cluster, which are assigned MPI ranks 8-23. The first compute node is Node#132 (MPI rank 8). The gateway agent itself has a simulated MPI rank of –3, and could have pointed to an additional three clusters/gateways if needed.

Node #9, the right-child of the root sentinel agent, is the first local/public compute node and the ancestor of all the other public compute nodes. The public compute nodes are assigned MPI ranks after all the remote-cluster compute nodes. So in this example, Node#9 gets MPI rank 38, and its descendants get the remaining MPI ranks.

NOTE that this tree structure results in certain restrictions on agent relocation:

- Cluster compute nodes can move around within a cluster, but cannot easily move to other clusters or to public compute nodes. This would require their (and also other nodes') MPI ranks and Node-ID's to change, since a compute node's MPI rank and Node-ID is determined by its location within the logical tree structure, and performance requirements discourage logical clusters from being split over multiple physical clusters.
- Similarly, public compute nodes cannot be relocated to remote clusters.
- If a cluster gateway dies, causing all the compute nodes within that cluster to be abandoned, the entire cluster (gateway and compute nodes) must be relocated to another available cluster that is at least as large as the original cluster, in order to keep the cluster numbers and MPI rank numbers the same.

Overview of the SentinelAgent Argument Structure

The args[] array is created by the Commander agent in spawnSentinel() after extracting cmdline parameters and querying the Resource agent, and is then passed to each Sentinel agent. The sentinel agents then use it to create the agent tree structure above. The following table gives the format of the array (the entries in **red** are changes from the design spec):

Arguments	**Remarks**
args[0]	#computing nodes, (i.e., MPI.size( )):
args[1]	#cluster systems: Let it be C.
args[2]	Cluster 1's name
args[3]	Cluster 2's name
…	…
args[C + 1]	Cluster C's name
args[C + 2]	Cluster 1's #nodes: Let it be C1.N
args[C + 3]	Cluster 2's #nodes: Let it be C2.N
…	…
args[2C + 1]	Cluster C's #nodes: Let it be CC.N
args[2C + 2]	Cluster 1's gateway alias or name
args[2C + 3]	Cluster 1's computing node[0]
…	…
args[2C + C1.N + 2]	Cluster 1's computing node[C1.N – 1]
args[2C + C1.N + 3]	Cluster 2's gateway alias or name
args[2C + C1.N + 4]	Cluster 2's computing node[0]
…	…
args[2C + C1.N + C2.N + 3]	Cluster 2's computing node[C2.N – 1]
…	…
args[2C + 1 + $\Sigma_{i=1}^{C-1}$(Ci.N + 1)]	Cluster C's gateway alias or name
…	Cluster C's computing node[0]
…	…
args[2C + 1 + $\Sigma_{i=1}^{C}$(Ci.N + 1)]	Cluster C's computing node[CC.N – 1]
args[2C + 2 + $\Sigma_{i=1}^{C}$(Ci.N + 1)]	#single desktop machine names: Let it be S.
args[2C + 3 + $\Sigma_{i=1}^{C}$(Ci.N + 1)]	Desktop 1's name
args[2C + 4 + $\Sigma_{i=1}^{C}$(Ci.N + 1)]	Desktop 2's name
…	…
args[2C + S + 2 + $\Sigma_{i=1}^{C}$(Ci.N + 1)]	Desktop S's name
args[2C + S + 3 + $\Sigma_{i=1}^{C}$(Ci.N + 1)]	**Root sentinel's machine name**
args[2C + S + 4 + $\Sigma_{i=1}^{C}$(Ci.N + 1)]	**#of bookkeeper agents**
args[2C + S + 5 + $\Sigma_{i=1}^{C}$(Ci.N + 1)]	User application's program name
args[2C + S + 6 + $\Sigma_{i=1}^{C}$(Ci.N + 1)]	The application's argument 0
args[2C + S + 7 + $\Sigma_{i=1}^{C}$(Ci.N + 1)]	The application's argument 1
…	…
args[2C + S + A + 5 + $\Sigma_{i=1}^{C}$(Ci.N + 1)]	The application's argument ***A*** – 1
args[2C + S + A + 6 + $\Sigma_{i=1}^{C}$(Ci.N + 1)]	"_?end_of_user?_"
args[2C + S + A + 7 + $\Sigma_{i=1}^{C}$(Ci.N + 1)]	Extra clusters (as in args[1] ~ args[2C + 1 + $\Sigma_{i=1}^{C}$(Ci.N + 1)])
…	Extra desktops (as in args[2C + 2 + $\Sigma_{i=1}^{C}$(Ci.N + 1)] ~ args[2C + S + 2 + $\Sigma_{i=1}^{C}$(Ci.N + 1)]

(Adapted from Prof. Fukuda's InterClusterDeployment document)

The following table shows how the args[] array might actually look, in the example of the sentinel tree above:

args[0]	#computing nodes: 43
args[1]	#cluster systems: 4
args[2]	Cluster 1's name: C1
args[3]	Cluster 2's name: C2
args[4]	Cluster 3's name: C3
args[5]	Cluster 4's name: C4
args[6]	Cluster 1's #nodes: 8
args[7]	Cluster 2's #nodes: 16
args[8]	Cluster 3's #nodes: 4
args[9]	Cluster 4's #nodes: 10
args[10]	Cluster 1's gateway alias: C1-alias
args[11]	Mnode0
args[12]	Mnode1
…	…
args[18]	Mnode7
args[19]	Cluster 2's gateway alias: C2
args[20]	Slave0
…	…
args[52]	#single desktop machines: 5
args[53]	Desktop 1's name: D0
args[54]	Desktop 1's name: D1
args[55]	Desktop 1's name: D2
args[56]	Desktop 1's name: D3
args[57]	Desktop 1's name: D4
args[58]	Medusa
args[59]	1
args[60]	User application's program name
…	…

(Adapted from Prof. Fukuda's InterClusterDeployment document)

The current args[] array structure is an adaptation of the earlier args array used before inter-cluster deployment was developed. In its current state, it has a number of weaknesses:

- It is difficult to find specific array elements, though there are some utility functions in AgentUtil.java to assist with this.
- It results in code that is difficult to maintain, with many "off-by-one" bugs and such whenever the array format is modified.
- It uses a sequentially structured format that does not lend itself to efficient random access.
- It is not updated as agent relocations take place over the course of the compute job, thus after the initial job launch is completed it must be used very cautiously. In general, the code assumes that the **cluster order** and **#nodes-per-cluster** will remain the same, however the actual compute nodes and clusters can be relocated (subject to the restrictions listed above).

The structure is normally not modified after startup, except for one notable case: Extra-clusters which are in use are marked as such (by setting the cluster-node-count to 0). This is done in place of using an duplicate nextNodes list for clusters.

Major Changes to CommanderAgent.java

1. New launch parameters to support bookkeepers and extra clusters:

    RB_<NumBookkeepers> - Specified the #of bookkeeper agents to launch.

    ECL_cluster_gateway_ipname{[_cluster_node_ipname]} - Specifies extra remote clusters to use.
    The cluster gateway name comes first, followed by the cluster gateway node alias, followed by a list of machine nodes within that cluster. If it is not given, a resource agent is responsible to provide the commander with such a list.
    To specify extra-nodes for a regular cluster, use an ECL parameter that has the same cluster name as the regular cluster.

2. The CommanderAgent **constructor** was enhanced to support the new launch parameters.

3. **spawnSentinel()** was modified to populate the args[] array with the extra-cluster information.

4. **spawnResource()** was modified to send our resource request to the new resource agent.

4. **receiveResourceItinerary()** was rewritten to receive and process the resource agent's allocation response.

Major Changes to SentinelAgent.java

1. **initArgs()** was modified to populate the nextHosts list appropriately (it had previously been completely rewritten to support remote-cluster deployment). Cluster-gateway sentinels and cluster-node sentinels use extra-nodes within their own cluster, while other sentinels use public extra-nodes.

2. **resume()** was rewritten to handle cluster-gateway and cluster-node resumption. It uses the following logic for these two types of sentinels:

a) Determine what cluster the sentinel is now in.
b) Is it different than the one it was previously in? If no, no special action is needed. If yes, then do the following:
c) Get the existing cluster compute-node-count.
d) Reserve the correct #of nodes for the cluster compute-nodes, and put the rest in the cluster's extra-nodes list (the new cluster may have more nodes than we need, in which case the excess nodes are reserved for job resumption).
e) Allocate specific nodes for our immediate children. Because we have a large number of descendent nodes restarting in parallel here, I don't trust the normal distributed nextNode allocation logic, and so I instruct AgentUtil.retrieveAgent() to use specific computers.
f) The existing (mostly) dead-child detection logic is then relied on to automatically restart the rest of the cluster nodes.

3. **sendSnapshot()** was modified to communicate with the correct bookkeeper agent, via the new AgentUtil.mapSentinelRankToBookkeeperId() function.

Major Changes to AgentUtil.java

1. New member variables added to store sentinel args[] array (if a sentinel agent), and the nextNode override list used for resuming sentinel nodes.

2. New **setSentinelArgs()** property added, to set the sentinel args variable above.

3. **retrieveAgent()** modified to use the nextNode override list mentioned above, which stores NodeID+MachineName pairs. RetrieveAgent() first looks in this override list to see if the NodeID to be resumed has been assigned to a specific machine. If so, it resumes the agent on that machine and then removes the override entry from the override list (it's a one-time-use thing). If no override is found, then the existing nextNodes logic is used.

4. **pingToChild()** modified to handle dead cluster gateway sentinels, using the following logic:

a) Is the dead child a sentinel gateway?
b) If YES, then restart the child at the next available cluster of sufficient size. First find an unused spare cluster of sufficient size. Then tell the resume logic to use this cluster's gateway for the gateway sentinel rather than the normal nextHosts list. Once the cluster gateway is started in the new cluster, the new resume() logic in SentinelAgent above will ensure the cluster compute nodes are also resumed on the correct machines.
c) If NO, then restart the child normally using nextHosts list (subject to any nextNode-override-list entries of course).

5. **pingFromParent()** modified to handle dead cluster gateway sentinels, using the following logic:

a) Is the dead parent a sentinel gateway?
b) If NO, then restart the parent normally using nextHosts list.
c) If YES and we're a cluster compute-node, then call **killAgents()** to kill us and all our children, since the cluster is now officially a dead cluster.
d) If YES and we're a cluster gateway, then restart the parent gateway sentinel using the same logic as with child gateway sentinels above.

6. **restartUserProg()** probably also needs to be modified to mark used extra-clusters in the same way that it marks used extra-nodes.

7. There's a number of new functions used by both intercluster deployment (from my preceeding CSS499 project), by intercluster job resumption, and/or by intercluster bookkeeper access:

calculateTreeSide() – Determines whether a NodeID is a public node, a cluster, or the root sentinel.
calculateInnerLayer() – Calculates the inner layer, that is, the layer within a cluster.
calculateOffsetFromLeftMostAgent() – Calculates the agent's offset within its rank.
calculateClusterGatewayNodeId() – Calculates the cluster gateway sentinel for this cluster node.
calculateClusterNumber() – Calculates 0-based cluster# from cluster gateway id.
isGatewaySentinel() – Returns cluster# if this agent is a gateway sentinel, or -1 if not.
calculateClusterStartingRank() – Calculates the starting MPI rank of this cluster.
calculateAgentRankWithinCluster() – Calculates relative MPI rank within a cluster.
calculateAgentRankWithinLocalNodes() – Calculates relative MPI rank within public computes nodes.
calculateSentinelRankFromId() – Calculates sentinel MPI-rank from sentinel-id.
mapSentinelIdToBookkeeperId() – Calculates the bookkeeper id corresponding to a given sentinel id

mapSentinelRankToBookkeeperId() – Maps the Sentinel mpi-rank to the corresponding Bookkeeper mpi-rank.

8. A number of utility functions that operate on the sentinel args[] array:

getClusterCount() – Gets number of clusters.
getComputeNodeCount() – Gets number of compute nodes.
getClusterNodeCount() – Calculates the number of nodes in this cluster.
getClusterDetailOffset() – Gets the offset in the args[] array where this cluster's details begin.
getLocalNodesOffset() – Gets the offset in the args[] array where the public compute-nodes begin.
getRootSentinelOffset() – Gets the offset in the args[] array where the root sentinel is located.
getUserAppOffset() – Gets the offset in the args[] array where the user-app information begins.
getUserAppArgCount() – Gets the user-app argument count.
getExtraClustersOffset() – Gets the offset in the args[] array where the extra-clusters begin.
getExtraClusterCount() – Gets the number of extra-clusters (note that some of these are extra-nodes for regular clusters).
getExtraClusterDetailOffset() – Gets the offset in the args[] array where this extra-cluster's details begin.
getExtraClusterNodeCount() – Calculates the number of nodes in this cluster.
getExtraLocalNodesOffset() – Gets the offset in the args[] array where the extra-nodes begin.
findExtraClusterBySize() – Finds an extra-cluster that has a specified minimum size.
findExtraClusterByName() – Finds the extra-cluster with the specified cluster name.
findExtraClusterByGatewayName() – Finds the extra-cluster with the specified cluster gateway name/alias.
findExtraClusterByNodeName() – Finds the extra-cluster that contains the specified compute-node name.
markExtraClusterAsUsed() – Marks this extra-cluster as being in use.