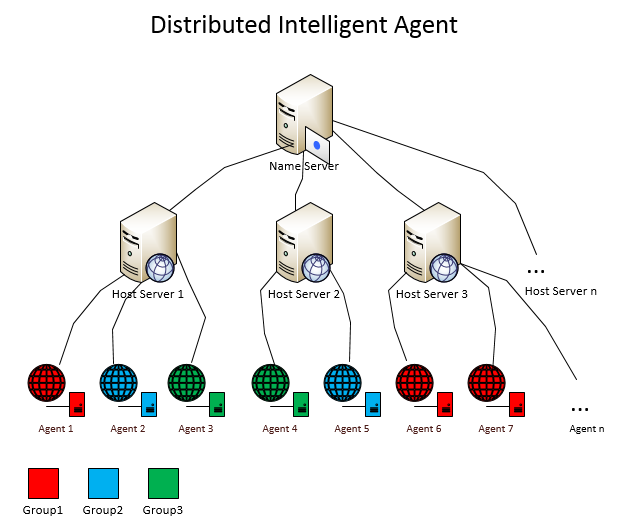
Distributed Intelligent Agent

1. Requirements

Build a distributed web server. This web server contains multiple host servers. Each server can holds several agents. Each agent can handle the client requests at specific ports. These agents can be assigned to different groups. The agents within the same group can share their data and state. They may also be able to communicate with each other. Moreover, there is an independent name server, which maintains the entire status of this web server, including the host server, agents, and their states.



1. Technical Design
   1. Data Model
      1. Server(Entity class for host server)

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Properties** | **Type** | **Description** |
| 1 | IP | String | Host Server IP |
| 2 | Port | Int | Host Server Port |
| 3 | Agents | List<Agent> | Agents held by this server |

* + 1. Agent(Entity class for agent)

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Properties** | **Type** | **Description** |
| 1 | ID | Int | Agent id |
| 2 | Name | String | Agent name |
| 3 | IP | String | Agent IP |
| 4 | Port | Int | Agent Port |
| 5 | Server | Server | Where the agent locates |
| 6 | Group | Group | Belongs to which group |

* + 1. Group(Entity class for group)

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Properties** | **Type** | **Description** |
| 1 | ID | Int | Group id |
| 2 | Name | String | Group name |
| 3 | Color | String | Color |
| 4 | Port | Int | Port for UDP broadcast |

* 1. Name Server
     1. Start up

Name server shall be started up before other host servers and agents being setup. There should be only one name server for DIA system. The instance of the name server contains three list for host server, agent and group. Group list is generated along with the creation of name server. It contains three items, each has a specific color, which makes easy to distinguish the agents later.

* + 1. Monitor

After the Name server is setup, it’s ready to handle requests. There are two channels that the name server is monitoring at: one is for handling the request from client browser which asking for the latest state of the DIA server; another is for processing the communication from other host servers or agents. For the first one, name server will setup a socket which listens at **port 48060.** When a request comes, it will get the latest information of host servers and agent, then generate html page and send back via TCP. For the second one, name server will start a CommunicationListener, which runs in a separate thread, to handle the request.

* 1. CommunicationListener
     1. Start up

A communication listener is started up when the name server is generated. It has the reference to the host serve list, agent list and group list, which are owned by name server. The communication listener will manipulate these lists when receives UDP messages.

* + 1. Monitor

The communication listener monitors at **port 48050**. It will receive the UDP messages from host servers or agents, handle it, sometime send result back to the sender. Each message contains a header to indicate what request it is, so the listener can handle it accordingly. The requests are listed as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Message Header** | **Description** | **Feedback** |
| 1 | [NewHostServer] | Register new host server |  |
| 2 | [NewAgent] | Register new agent | Send back agent name |
| 3 | [RequireHostServer] | Look for available host server for migration | Send back host server’s ip and port |
| 4 | [AgentMigration] | Update agent’s new ip and port after migration |  |
| 5 | [AgentKilled] | Remove agent from list after being killed |  |

* 1. Host Server
     1. Start up

Host server shall be started up after name server has been setup. Unlike the name server, there can be multiple host servers within one DIA system. Host server can be started up with specified port number, or with **default port 45050**. Each instance of the host server contains an agent list, which means one host server can create multiple agents. Once the host server is created, it should contact name server to register itself.

* + 1. Monitor

Host server monitors at the specified port or 45050 by default. Whenever a request comes, it will create a new agent for it.

* 1. Agent
     1. Start up

The agent is the instance of class AgentListener, which is created by host server. Each agent will find a new available port number when it is created. And it will contact name server for registration.

* + 1. Monitor

Agent monitors at its port to handle requests. It dispatches the requests to AgentWorker.

* 1. AgentWorker
     1. Start up

A new instance of AgentWorker will be created each time when a new request is received by agent.

* + 1. Handle Requests

The requests that a worker handles are listed as follows:

|  |  |  |
| --- | --- | --- |
| **No.** | **Action Type** | **Description** |
| 1 | Refresh | Update the state (each time increment 1) |
| 2 | Migration | Migrate agent to another host server and port |
| 3 | Kill | Kill agent |

* 1. Log

One requirement is that the execution logs can be saved to log file. This LogHelper class is for this purpose. It contains three static methods, see below.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | Write() | Add log to the list |
| 2 | Save() | Save the list to file, one item one line |
| 3 | Clear() | Clear log list |

* The logs for name server are stored in file **Log\_NameServer.txt**.
* The logs for host server and its agents are stored in file **Log\_HostServer\_#port#.txt**.
  1. UdpHelper

UDP communication is used in name server, host server and agent. It is necessary to create a common helper class to provide the basic udp operation services. Class UdpHelper is invented for this purpose, it contains two following methods.

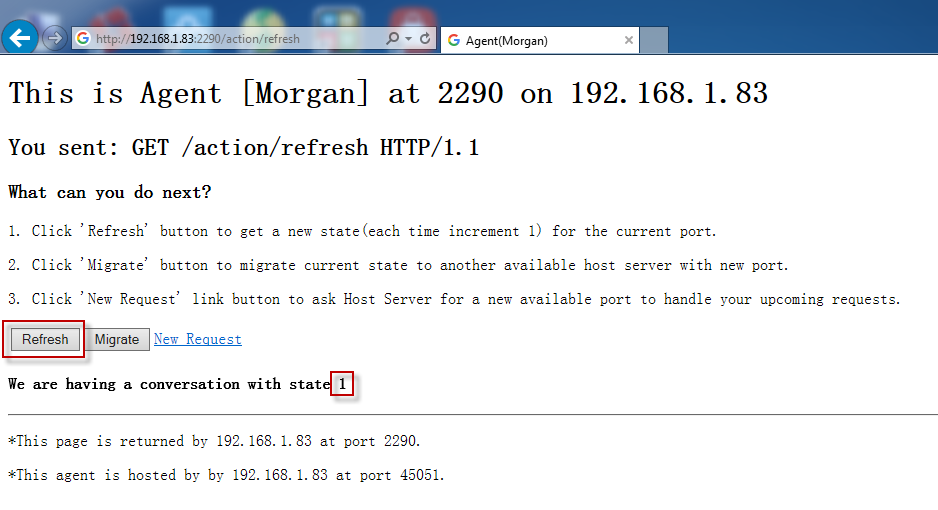
|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | sendAndReceiveUDP() | Two way operation, sending and receiving. |
| 2 | sendUDP() | One way operation, only sending. |

1. Functionality Discussion
   1. State Update
      1. Requirements

Agent can provide appropriate response to different kinds of request. No matter how agent handles them, it always shall be able to maintain its state. To make it simple, we assign an integer value to each agent as the ‘state’. Each time the agent is accessed, the state will be returned. And this number can be increased if user want to update it.

* + 1. Implementation

We create a button in the page, named ‘Refreash’. Each time it is clicked, browser will send an HTTP GET request to agent.



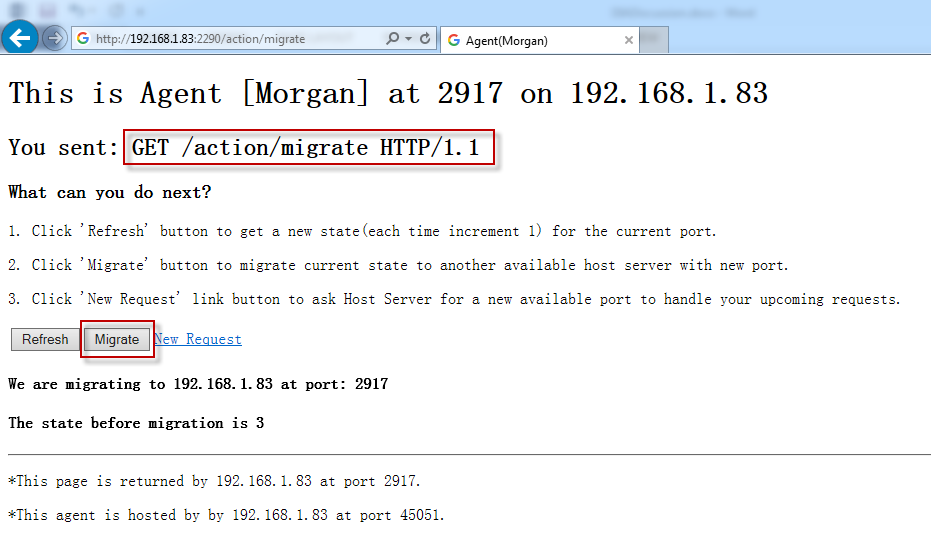
Agent Listener will get this request, something like ‘GET /action/refresh HTTP/1.1’. It will assign this request to Agent Worker. Agent Worker will increment the number by 1 and send the new value back to client.

* 1. Migration
     1. Requirements

Agent can migrate to other host server and port with its states remaining.

* + 1. Implementation

We create a ‘Migrate’ button in the page. When user click on it, a migration request will be sent to Agent.



Agent Listener will get this request, for example ‘GET /action/migrate HTTP/1.1’. It will assign this request to Agent Worker. The whole procedure is described as follows:

1. Agent Worker contacts name server to ask for a new available host server. It setups a UDP connection at **port 48050**, which the name server is listening to.
2. And the worker creates a message with header ‘[RequireHostServer]’, then sends it out.
3. The name server gets the request, finds an available host server from its server list, then, send back the IP and port of the new host server.
4. Agent worker gets the IP and port, setup a TCP connection to the new host server to ask for a new valid port for migration.
5. Host server gets the request, creates a new agent. This new agent sends information to notify the old agent.
6. The old agent reads the response, gets the new port. Then, build the http response with current states, send to client browser to inform end user that the agent has been migrated. At last, the old agent shutdowns itself.
7. The new agent also sends a UDP message to name server to let it update the agent information.
8. Name server gets the update request, updates the agent with new port.

Note that, after migration, the agent name and states are still same. The port must be changed.

Besides, here in my implementation, only a little piece of data are transferred to the new agent. If necessary, we can transfer all of the states, including ID number, group, server, data, history, etc to the new agent along with the migration. There are two ways to do that: a). Combine all of information to a long string, send to the new agent and let it to parse and rebuild them. b) Make the model class(Server, agent, group, etc) implement ‘Serializable’ interface, directly send the objects to new agent. The new agent just needs to de-serialize them.

* + 1. More

In my implementation, I just kill the old agent. Actually, we can improve the design about the old agent. We can keep the old agent alive. However, it doesn’t work as normal before migration.

1. We create a new property for the agent, to indicate whether it is a zombie. The default value of it is false, it becomes to be true after the agent has been migrated. After migration, it becomes a zombie, and it has the information about the new agent. It still monitors the old port.
2. When a new request comes to the zombie, it won’t print any html response to the client. Instead, it send back the http 302 status code and the URL of the new agent. Sample like this:

**HTTP/1.1 302 Found**

**Location: http://localhost:2234/**

Then end user’s browser will be redirected to the new agent directly.

1. If there are some GET parameters in the original submission, we can extract and add them to the redirection URL.
2. If there are some POST parameters, we can also build the redirection via http1.1 status code 307 (Temporary Redirect).
   1. Agent Communication

I have not implemented the communication between agents. Here are my thoughts about this issue.

* + 1. Setup

Each agent is assigned a group when it is created. When a new agent is added to the group, name server will inform other agents within the same group. Since name server has the whole knowledge about the system, it can find all of the agents in the same group. Then, it can setup UDP connection to each of them, and send them with the latest group data. Each group has a specific broadcast port, name server just need to send latest data to this port via UDP message.

* + 1. DIA Table

Name server needs to maintain a table for storing the Label/Description.

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Properties** | **Type** | **Description** |
| 1 | ID | Int | Identity |
| 2 | Label | String | Label |
| 3 | Description | String | Description |
| 4 | IsLocked | Boolean | Flag, true is locked by someone |
| 5 | LockedBy | Int | Agent id |
| 6 | LockTimeout | DataTime | Timeout for the lock |

* + 1. New Label

Agent sends ‘New Label’ request to name server at communication port 48050 along with the agent id, group id. The name server gets the request, search it in this table.

* If no found, insert it, add lock and set timeout(30 minutes for example). Send acknowledge to the agent, and agent can work on it.
* If found, send response to the agent along with description, tell that it is already existed. Agent will see the error message and the description of the label.
  + 1. Check New Label

End user can input any text as label to search whether it exists. Agent sends ‘Check New Label’ request to name server at communication port 48050 along with the agent id, group id. The name server gets the request, search it in this table.

* If not found, send error message to the agent.
* If found, send response to the agent along with the description so the user will see it.
  + 1. Get Matching Data

End user sees label in the page, and request for the description of it. Agent sends ‘Get matching data’ request to name server at communication port 48050 along with the agent id, group id. The name server gets the request, search the label in DIA table.

* If found, send the description to client and the agent will see it.
* If not found, send error message to the agent, the agent will know the matching data does not exist.
  + 1. Submit Description

End user tries to submit a description for a label. Agent sends ‘Submit matching data’ request to name server at communication port 48050 along with the label id and agent id. The name server gets the request, search the label id in DIA table.

* If no found, send error message to the agent and the submission can’t continue.
* If found, check whether it is locked and whether the lock has expired.
* If there is a valid lock, send error message to the agent and agent can chooses to wait or give up the submission.
* If the no lock or the lock is expired, then add a lock for this agent first, and inform agent can continue.
* Agent is notified by the name server that it has gained the privilege. Then it waits end user to edit the text and send to the name server.
* The name server receives the agent id, label id and description, notices that this agent obtains the lock previously, so updates DIA table with the submitted description.
  1. Lunch Problem

In the ‘submit description’ operation, there is a **lunch problem** if a lock is applied. An agent may wait for a long time but still can’t get chance to update the description.

* + 1. Basic Solution

Our solution here is to add a timeout attribute to this lock.

* Whenever an agent try to submit description for the label, it must gain a lock(write right) first.
* In the first connection, the agent just ask for a write lock for continuous operation. The name server will assign it a lock, and set the expiration of the lock.
* Once the agent gets the lock, it inform end user to input some texts and submit the description in the second connection.
* If it doesn’t submit the description, it will lost the privilege after lock expires. And it has to apply again for submission description.
  + 1. Advanced Discussion
* We add the lock in entry level, not whole database. This is to reduce the lock scope as small as possible.
* If there are huge number of agents and labels, we need to consider to reduce the timeout duration. But still need to make sure end user has enough time to input the text.
* The bottleneck is in the name server, since it owns the DIA table and maintains the whole state. Moreover, all requests are sent to name server to process.
* The risk is when the name server goes down or collapsed, the whole DIA system can’t work.
  + 1. Improvements

There are several approaches to improve the performance and keep the system robust.

* Create a shadow server for name server. Name server persists data to disk, and writes log to disk for each operation. When it goes down because of system failure or exception, the shadow server can replace the name server immediately. It loads the data from db file first, and read the log then. Recreate the operations mentioned in the log. It becomes a new name server.
* Separate the db file to small chunks, for example, by alphabet. Create 26 db files for each letter. All labels with same first letter will be put into the same file.
* Create workers for the name server. Each worker only focuses on a small range of the requests. For example, worker A only handles tasks from letter ‘A’ to ‘D’, worker B handles ‘E’ to ‘G’, ….
* Create dispatcher for the name server. The dispatcher does nothing but assign tasks to appropriate workers.
* Finally, the pressure of the system are cut to small pieces, no bottleneck anymore.
  1. Sharing Data

There are lots of communication between agents and name servers. The problem here is, how to make the communication more efficient.

* + 1. Where the data stores?

For agent, it must maintain the information about itself, and the data which used to communicated with other agents and name server. For the name server, since it is central server, the server list and agent list must be maintained. There should be some redundancies of the data between these agents and servers.

* + 1. Synchronization

The critical issue is how to synchronize the data if some changes are happened. If there are only a few agents and server, this is not a big problem. However, if there are thousands or millions in the same DIA system, it is hard to synchronize quickly and the cost for synchronization is very high. Generally, there are two ways of synchronization, pull and push.

* + 1. Poll

Poll means each time I want to get new updates I need to access the resource and poll the information by myself. The problem for this approach is, there may be no updates at all. But I have no way to know that, so I have to keep polling even nothing new is found. So, there are some wasteful communications for the whole system.

* + 1. Push

Push means whenever a change occurs, a publisher will push the changes to the subscriber immediately. The benefits is that the subscriber can get the notification in time. There are also some problems with this approach. Suppose, there are huge numbers of the subscribers, it may takes long time to notify every one of them. It may affect the performance of the publisher. And it cannot make promise that every subscriber get the update at the same time. Sometimes, this will bring consequent issues. Besides, in the reality, not all of the subscribers need every notification of the change. There are also some wasteful communication for this approach.

* + 1. How to choose?

We may combine these two approaches in our DIA system.

* We let the host server hold the data of agents and DIA table.
* Whenever any change to agents and DIA table, the name server will update first. Then it use the push approach to push the changes to host server. Generally, there are only few host servers within a DIA system. So the performance should be ok, and the communication cost won’t be high.
* Then, whenever an agent wants to get information, it use the poll approach to ask the host server where it locates for the data.
* For the group’s data sharing, if it crosses several different host server, we can setup another server to maintain these group data. Similarly, it can get data from name server via push approach and share data to agent via poll approach.
* Finally, we reduce the communication between agents and servers.