Randy Armknecht
DePaul University
SE 435 - June 2008

# On the Design of a Distributed Intelligent Agent for Playing "The Animal Game"

By its very definition a Distributed Intelligent Agent (DIA) is a system of multiple programs (agents) that work together to expand the system's collective knowledge (intelligence) located on one or more physical machines throughout the world (distributed). For the purposes of this paper the design objectives of the system include the ability to be geographically dispersed, resilient in maintaining knowledge despite unpredictable loss of servers, composed of agents with the autonomous ability to migrate both within and between servers, for agents to belong to groups, and for the system to be able to simultaneously play the animal game with multiple clients resulting in a more comprehensive knowledge base. The path to meeting these design objectives is fraught with many tough choices each with their own tradeoffs. Below, the problems exposed and the solutions chosen are examined in detail.

Before beginning it is important to understand how the animal game is played and the overall goal of the system. The game is played by a client first thinking of an animal followed by a series of yes or no questions posed by the agent. The agent uses the client's responses to form an educated guess about which animal the client is thinking of. The agent uses the results of each question and the final result to gain knowledge. Knowledge could be learning a new animal, or learning more facts about existing animals. The goal of the system is to learn as many facts about as many animals as possible, so as to never be stumped by the human client.

## PROBLEMS

The first problem faced in the design of this system is the determination of what parts *exactly* are required to make a minimum complete system. Its easy to overlook this step and jump into the more interesting problems, but it absolutely cannot be skipped. Without a clear idea of what the system is, choices will be made without the fullest of information available, resulting in a less than optimal system.

The most basic system consists of a host server (HS) with multiple agents on it. Clients interact with the system by connecting to the HS on a specified control port which triggers the generation of an agent for that specific client. All further communication between the client and the system are facilitated by the agent. The system is capable of growth in two directions; first, by creating more agents on the host server, and second, by joining additional host servers with their own agents into the system.

One problem that must be addressed for this system of multiple agents located on multiple servers is how each is referred to. The obvious and most familiar solution would be to refer to each by a unique name. So then how is this name generated in such a way as to guarantee uniqueness? Furthermore, how will the system find other servers and agents? Will it require the storage of a complete list of names and network locations on each node or can it be implemented in some type of location service? If a location service is used, where is it located, how it is known to the system, and what happens if it fails? These issues are just the start of the ones faced in the design of a complete system.

Another issue is effectively sharing the collaborative knowledge of the system with all the nodes. Nodes may exist in different time zones across the world, will this play a role? Will the knowledge be stored on each agent? Each group of agents? Each host server? In a centralized location for the entirety of the system? If it's stored in multiple locations is a distributed transaction system required?

Once the knowledge has been stored effectively, how is it accessed and utilized by the agents? What data structure is it stored in? How does that structure get effectively transferred across the network? What protocols are used? And then once it's transfered to an agent, how can it update the collective knowledge with what it just learned from a client? How is this new knowledge shared between agents? Do they message each other with updates, or do they rely on a central location?

With all these problems raised about the storage, transfer, and interaction of the systems knowledge it should be noted that a very important question hasn't yet been asked. That is, how important is it that the agent has the most recent knowledge? Can an agent still benefit the client and the system if it's knowledge is out dated? If so, for how long? If an answer can be provided to these last three questions the solutions to the previous questions can be narrowed down and chosen more wisely.

Agents belong to groups. The group memberships must be maintained somewhere, so where is that? Are groups local to a single Host Server? Can group membership span several host servers? How is an agent's group chosen? If an agent choses to migrate from one host server to another, does it stay in the same group? What if no other agent on that server is part of the same group? And how exactly does an agent decide to migrate?

A couple more problems that need to be considered are: when a system is up and running how does another host server get added into it? And what does an agent do if a client just stops responding, but has maintained network connectivity? (i.e. the lunch problem).

Finally, the types of transparency provided by the system need to be decided on. Designing solutions to the problems encountered so far hinge on types of transparency that must be adhered to. Additionally, the common pitfalls outlined on page 16 of *Distributed Systems: Principles and Paradigms* need to be reviewed so that the design can avoid these.

It's been shown that every decision faced in the design of this system will require some type of trade off (complexity for redundancy, centralized server for simplicity, etc). In the next section each of the above listed problems will be considered and a design decision made.

## SOLUTIONS

In an effort to better organize the solution to this unique challenge the design choices will be presented across three sections; host server design, agent design, and over all system design.

### HOST SERVER DESIGN

Since each agent within the system resides on a host server (HS) and there may be multiple HSs within the system, there must be a unique name for each agent and server. The servers are easier to name than the agents within the system since they do not migrate and are less likely to fail. Each server connecting to the internet for participation in the system must already have a unique IP address and port number. This information will be used as the name of each server. The port is required since a company or individual may only own one public IP address and desire to host multiple HSs behind it using a technology such as Network Address Port Translation [1]. This design choice will not prevent them from doing so and adds relatively little in complexity to the code base.

One of the overall design goals is to provide for massive scalability. It is therefore an immediate thought to implement a distributed transaction system such as two-phase commit for the replication of important data (servers, groups, knowledge, etc) across many nodes. [2] Unfortunately, the implementation of that is quite complicated and provides guaranteed consistency across nodes. That guarantee is not required, as will see later. So in an effort to keep things orderly and simple the implementation of a distributed transaction system as been avoided. Instead, there will be a centralized coordinator for the system.

Initially, the coordinator will be the first HS to create the system. That is, once a HS is executed it can optionally be passed the IP:Port of any current HS in an existing system. If no IP:Port is provided, the HS will create a new system with itself as coordinator. When given an IP:Port the new HS will ask the provided HS for the IP:Port of the current coordinator. Once provided it informs the coordinator that it wants to join and is placed on the central list of active servers. It then becomes apparent that each HS stores at a minimum the IP:Port of the coordinator.

Above it was stated that failure resilience was a top priority. Again, with a focus on the large scale deployment of the system a failure of a HS other than the coordinator is an unfortunate event, but does no harm to the system as a whole and is deemed acceptable. The failure causes no harm to the system as it will only contain a small fraction of the system's total agents. While it's down, all other agents will be able to continue interacting with clients and providing new knowledge to the system.

Failure of the coordinator however is a much larger issue. The chosen solution for this is a combination of replication, super nodes, and an election algorithm. Since each server must register to join, the coordinator will have a complete list of all servers in the system. As

servers are added to the system, the coordinator will designate some as super nodes. These nodes will be provided replicated data from the coordinator, and should the coordinator fail, will participate in the ring algorithm [3] to elect a new coordinator from the super nodes. To provide for scalability, once super nodes are part of the system the coordinator only maintains a list of super nodes. Each server that isn't a super node will be assigned one when it registers with the coordinator. Thus each super node will have a list of all of the host servers assigned to it.

The ring algorithm starts when a super node realizes that the coordinator is offline. It will look at it's list of fellow super nodes and send an *ELECTION* message to the first IP:Port that is greater than itself and attaches it's own IP:Port. If that super node cannot be reached it tries the next on the list. The receiving super node will take the message, append it's own IP:Port and pass it on. If it's the highest on the list it will pass it to the lowest on the list. This repeats until the message arrives at the original sender. The sender changes the message to *COORDINATOR* and forwards it to the next super node on the list after itself. When the message is received the current locally stored list of super nodes is discarded and replaced with the list of IPs and ports from the message, the lowest IP:Port on the list is marked as the new coordinator. This is repeated until the message arrives at the originator at which point it is discarded. This algorithm (fully explained on pg. 266) is a clean and simple way to provide failure redundancy for the coordinator without implementing a distributed transaction system. Additionally, it provides for a more hierarchal approach to the overall solution.

It should also be noted that when a super node is created, other host servers are informed that they should report to the new super node. The fact that there is a coordinator above the super node is irrelevant. A super node treats all assigned nodes as its own "super nodes", replicating to them the list of nodes in the group and the list of all other super nodes. Should a super node fail, the election algorithm will take place to elect a new super node. With this in place a host server should never end up isolated without all of it's peer nodes and all the super nodes simultaneously failing.

With the segmented host server nodes greater redundancy has been introduced, but at what cost? First off, it's known that agents will migrate both within a server by changing ports and across servers. It's possible for an agent on HostServerA to migrate to SuperNodeA then to SuperNodeB and then again to HostServerB which is under SuperNodeB. How could this agent be located? The ability to locate the agent could be implemented by using a location service. As this is considered, it's a solution for a problem that doesn't exist. Since only one client at a time is communicating with an agent, it is trivial for the agent to migrate in the following manner: send current state to new host server and receive new port number if the migration was accepted, inform client of new location, destroy self. Since the current state was sent to another server, when the client connects it will still be in exactly the same state as before the migration, and the agent will have the same unique ID. In this instance it was a trade off between advanced migration capabilities with a location service and simple one hop migration capability without a location service.

## AGENT DESIGN

Each agent lives on a single host server and possess the ability to migrate from server to server one hop at a time. To effectively track a single instance of an agent as it moves throughout the system, it must have a unique name. While a creative scheme could possibly be imagined to ensure that every agent remained unique, it's often easier to stand on the shoulders of giants. Each agent upon it's creation will be assigned a Universally Unique Identifier (UUID)[4].

Now that agents can be easily referred to, they can be allowed to travel the system. Each agent has two potential ways in which to migrate: locally or remotely. If an agent is migrating locally it will simply change ports on the same host server. If an agent is migrating remotely, it will move to a new host server and leave it's current server. An agent will chose to migrate autonomously, that is, without interaction of the client. The method for doing this is much like a dice roll. On each interaction with the client the agent will have a 1 in 20 chance of choosing to migrate. Unfortunately, it's unlikely but possible that an agent could migrate several times in quick succession. In order to prevent this a counter is set upon a successful migration to 10. If the counter has a value when a client interaction occurs the dice roll is not made, and the counter is decremented by 1. Only when the counter is 0 will a dice roll be made.

Each agent once created is automatically assigned to a group. The group name is the only name of a system component that will be seen by the client. In an effort to provide a friendly name to the client each group is named after a city throughout the world. This should be more pleasing to the client then a UUID. For the management of groups it is simplest to hard code a selection of cities into the base host server code. If enough default groups are included, say around 50, then it's less of a draw back than if only 2 default groups existed without the ability to create more. Upon creation, an agent will be assigned to a group randomly. This group name and any group attributes will be stored within the agent itself.

Members of a common group are able to communicate with each other only by suggesting the current color of the group. When a client tells an agent in group X to change colors, the agent will inform it's super node, which will inform the coordinator. The coordinator will inform all super nodes that the color for group X has changed via a version of the publisher-subscriber design pattern. [5] When a host server creates an agent that joins a new group that doesn't have any prior members on that server a subscribe request will be sent to it's super node. The super node will mark down the host server as a subscriber to that group. If the super node did not have the group it will be added and then subscribe itself to the that group's information with the coordinator. In this way an update from an agent will propagate in a method similar to multicast. The message goes from an originating agent, up to it's super node, up to the coordinator, then down to only the super nodes subscribed to that group, and then down from each of those to only the subscribed servers with agents in that group. If a server ends up with a group containing no agents due to migration or client termination it will send an unsubscribe request up to it's super node.

The last couple design issues with agents are the methods in which they interact with the clients, gain knowledge, and share that knowledge with the system. The interaction with the clients is web based. A client will connect to an administrative port on a host server which will then spin up an agent on a new port and use an HTTP 302 redirect to send the client to the listening agent. The agent displays a very simply web form that tells the user to think of an animal and offers a begin button. In the top center of the page the agent's group and a banner the color of the group is displayed. If the banner is clicked the user will get a set of radio buttons to chose a new color for the group.

Once the begin button has been selected the user will be asked a series of questions with yes or no answers. Each question is from a table of questions stored on the super node in a relational database such as MySQL. The questions asked are randomly selected from the database. After each question, the question ID, and the answer (yes or no) are stored in the agent. After 10 questions the agent will query the database for animals that match the provided truths and falsehoods of the questions asked thus far. The animal with the highest score (sum of matches for truth and falsehood) will be presented to the client. If correct, the agent will show a victory screen claiming it's greatness. If the agent was incorrect, it will ask 5 more questions and try again. If wrong still it will ask 5 more for a total of 20 questions. If still incorrect the agent will prompt the user for the animal's name. At this point the super node database can be updated to include the new animal name along with the 20 truths and falsehoods about it already provided by the client. If the client wants to, they can enter another question that would be true for the animal they were thinking of, but false for the animal last guessed.

Now that the agents under a specific super node are gaining knowledge by interacting with clients, it's easy to see that any information learned by an agent is immediately shared with all agents on all host servers reporting to the same super node. The concept of using MySQL was also arrived at through the philosophy of standing on the shoulders of giants. The database system has a robust and scaleable method for dealing with critical section locking and simultaneous transaction processing. [6] The super nodes will queue up their new knowledge over some period of time such as an hour. After this time they will submit their new knowledge to the coordinator. The coordinator will thank the super node by sending back knowledge gained after the super nodes previous contribution; knowledge that has been accumulated by all other agents and servers. The method for knowing when a super node last requested data can be as simple as a table on the coordinator that records an incrementing number with each transaction and the super node involved in it. This incrementing number can serve as a form of a Lamport Logical Clock. [7] The actual time on any given system is irrelevant and we'll always know which event happened before another event.

By only sharing knowledge with super nodes that have contributed knowledge the system is somewhat self policing. That is, if a rogue super node was able to join our system they could not gain knowledge from it without having contributed new knowledge to the system first. If the system administrators wanted, it could even be configured in such a way as to

enforce knowledge ratios. i.e. for every two pieces of knowledge provided three will be given back by the coordinator.

## OVERALL SYSTEM DESIGN

One overall design tradeoff that had to be made was the simple and scalable incomplete knowledge of each super node over absolute consistency and completeness of the systems data for every agent on every host server. This choice to have several separate realms of knowledge that can easily and periodically merge in a bandwidth friendly manner would not have been possible without the use of a relational database system. Not only does such a system avoid the problems inherent in merging binary trees, but it also avoids issues like the lunch problem, and increases the systems knowledge in a much quicker manner.

The lunch problem was briefly mentioned in the PROBLEMS section and occurs when a client walks away from the system and then returns at a later time. If a binary tree was used, that tree, or at the very least the current question node in use would have to be locked. The locked node could easily prevent other agents from utilizing it to improve the system's knowledge tree. With a transactional system such as MySQL, this issue will not occur as the table is not locked even if the client is in the middle of entering a new animal or question. The new information will simply be added to the database once the user has returned.

Another benefit of the database model is that much more knowledge can be gained about a new animal in a quicker time than if a binary tree was used. In the case of a binary tree, the client must enter a new question so that a new node can be created with the animal leafs moved around appropriately. In the database model, the user does not have to enter a new question and the system can still learn 20 facts about the animal. The improvement over the tree is that the questions can be much more numerous than 20, and that the questions of each game need not be asked in the same order to arrive at the same answer. It's possible that as the game gets played more and more, a single animal could be related to every question.

The system was designed to implement various types of transparency. For instance access transparency is implemented by hiding from the clients the complicated back end of the system. The client only knows that they pointed their web browser at a host server, and knows nothing of the databases, servers, and agents spread throughout the world. Location transparency is a by product of offering the system over the internet. Migration transparency is weakly provided. That is the user does not need to take any deliberate action to follow an agent when it migrates. However, HTTP redirects will be used, and to the observant client the URL will be changing as the agent migrates. Replication and Concurrency transparency are both provided through the super node / coordinator structure which encapsulates that from not just the majority of the host servers, but from the clients as well. Failure transparency is covered as far as the coordinator and the super nodes are concerned. If an agent dies, the client will notice. If a host server dies while multiple agents are in use, multiple clients will notice. The scope of clients impacted by such

a failure will be a small percentage of the whole. This is an acceptable level of failure transparency. Relocation is not used within the system. [8]

The system was also designed to avoid the 8 common pitfalls of distributed system design. First off, the network is not assumed to be reliable. Because of this the TCP protocol is used for all system communications. It has slightly higher overhead but will provide stronger connection reliability. Additionally, if a node cannot be reached, the system is designed to remove that node from the system. When the node comes back online it can rejoin. No assumptions are made about the security of the network, but from a practical standpoint, it doesn't need to be secure. Animal data is far from confidential and need not be protected. The network is not homogeneous, and the system does not depend on it being so. Nothing within the system is time critical, so the reality that a latency of zero does not exist will not negatively impact the system. Bandwidth is well known to not be infinite, which was one of the main reasons for building a hierarchal knowledge system and essentially multicasting group messages. Transport cost is also not zero, which is why multicasting is used for group messaging, and the database traffic is kept to small segments. The system also does not assume that there is only one administrator. In fact, the system assumes there are no administrators. Once a host server is joined into the system it becomes a cog in the wheel. Everything from node promotion to group multicast to coordinator election is autonomous. The only common pitfall the design may have fallen for is "the topology does not change." Host servers are referred to by their IP address and port number. If the network topology were to change, it's possible, but not absolute, that their IP address could change as well. If this happened, it would be disconnected from the system and treated as if it failed. Given that the animal game is not a critical system, this type of problem is acceptable. Also, the administrator of the network the host server was running on would know that the IP address was going to change and could re-add the server if desired after the change was complete. [9]

## CODE DESCRIPTION

It is important to note that the ideas proposed within this paper have not been implemented in actual running code by the author.

## SCALABILITY

This system should scale quite well due to the design decisions that have been made with scalability in mind. For instance, the concept of using super nodes within the system to form a three level hierarchy allows for communications to efficiently be passed throughout the system from one group member to other group members utilizing a method similar to multicast. Not every host server needs to get informed of group messages to be passed on to agents if it doesn't contain any agents in that group. The hierarchy also allows for loosely redundant knowledge to be efficiently shared amongst the whole. When one agent learns of a new animal it inserts it into the super node's database. With that one network message all other agents under that super node instantly have knowledge of the new

animal as well. The same applies to the update of the super node to/from the coordinator. With a single transaction all the servers under a super node gain knowledge of all the animals learned by all the other servers in the entire system.

The only issue with scalability that this system might have is that of the coordinator being bombarded by too many super nodes. It's possible that the super nodes could be spread too wide (i.e. one super node for every five servers) or that the system could simply become too big (i.e. one super node for every 1,000 servers, but 10,000,000 servers in all). If either of these occurred updates of the database to/from the coordinator might be too much to handle. Also, if the super nodes become too numerous the list of super nodes that they all share to utilize the ring algorithm could become too large to be practical. For instance, each item would take up 6 bytes (4 IP, 2 Port). If there are 100,000 super nodes each would have to have a 600k list. At a system that size it's practically guaranteed that some servers are on slow connections and transferring nearly a megabyte of data would tax the network in an unacceptable manner.

## SUMMARY

Designing a distributed system is much more involved and complicated than one may originally think. At each step throughout the design process a series of problems had to be addressed, each one with it's own tradeoff. The system designed above used a wide variety of already developed technologies from TCP and UUIDs to election algorithms and Lamport logical clocks to MySQL and super nodes. In the end they've been combined to form a hierarchal failure tolerant and scalable distributed intelligent agent system.

REFERENCES

1. RFC 2663 - IP Network Address Translator (NAT) Terminology and Considerations
   http://www.ietf.org/rfc/rfc2663.txt

2. Course Text Book - Section 8.5.1 pg. 355 - 360 "Distributed Commit Algorithms"

3. Course Text Book - Section 6.5.1 pg. 266 "The Ring Algorithm"

4. ITU-T Recommendation X.667
   Information technology – Open Systems Interconnection – Procedures for the operation of OSI Registration Authorities: Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 object identifier components
   http://www.itu.int/ITU-T/studygroups/com17/oid/X.667-E.pdf

5. Microsoft Patterns and Practices - "Publish/Subscribe"
   http://msdn.microsoft.com/en-us/library/ms978603.aspx

6. MySQL 5.0 Reference Manual - Section 7.3.1 "Internal Locking Methods"
   http://dev.mysql.com/doc/refman/5.0/en/internal-locking.html

7. Course Text Book - Section 6.2.1 pg. 244 - 248 "Lamport Logical Clocks"

8. Course Text Book - Section 1.2 pg. 12 "Transparency"

9. Course Text Book - Section 1.5 pg. 18 "Common Pitfalls"