Agent Coordination Using Enterprise Java Beans and XML Web Services in a Battlefield Environment

Bijan Zamanian
Department of Computer Engineering
Eastern Mediterranean University
Gazimağusa, T.R.N. Cyprus
bijan.zamanian@cc.emu.edu.tr

Zeki Bayram
Department of Computer Engineering
Eastern Mediterranean University
Gazimağusa, T.R.N. Cyprus
zeki.bayram@emu.edu.tr

Abstract

We describe an agent coordination architecture that makes use of XML Web services, Enterprise Java Beans, and lightweight threads of JAVA. The architecture is extendable and scalable due to the XML Web Services component – agents can exist, as objects running in processes on different machines, and yet live in the same virtual world. The proposed architecture is demonstrated in the simulation of a battlefield scenario, with agents of varying types, such as soldiers, tanks, planes etc. existing in two hostile armies.

1. Introduction

Complex tasks are often carried out by teams consisting of individuals, because no one individual has the collective expertise, information, or resources required for the effective completion or performance of a task. In the computer field, *agents* are used to model the characteristic behavior of individuals that are operating as part of teams, as well as their interaction with one another. *Autonomous software agents* are applications which are expected to accomplish their tasks using their skills and available knowledge in their operation domain.

Agents can cooperate to facilitate achieving a common, complicated and large scale goal. In such a case, each agent is responsible for achieving part of the goal. This cooperation effort has a chance of succeeding only by knowledge and information exchange and effective coordination [3][4].

Coordination is a process in which agents engage in order to ensure a community of individual agents acts in a coherent manner [1]. Coordination may require cooperation, but it is not the case that cooperation among a set of agents automatically results in coordination.

Competition and combat are forms of coordination in which agents in different camps try to defeat one another. Agents can cooperate and coordinate through communication by exposing their goals, results, statuses, threats and locations to each others [2].

In this paper, we describe an agent coordination architecture that makes use of XML Web services, Enterprise Java Beans, and lightweight threads of JAVA, and use our architecture to simulate a battlefield scenario involving two opposing sides. Our architecture is extendable and scalable because of the XML Web Services component – agents can exist, as objects running in processes on different machines, but operate in the same virtual world. Agent coordination is achieved through XML web services and Enterprise Java Beans.

In the battlefield simulation scenario, agents represent combat elements such as soldiers, tanks and planes. Each combat element initially asks for a mission from its command and control center, which is implemented as a set of Enterprise Java Beans (EJB). Each combat element, after obtaining its initial mission, sets out to accomplish it by moving in the direction of its target. Agents become aware of the environment and make changes to their environment using web services.

The remainder of this paper is organized as follows. In section 2 we describe the agent coordination architecture. In section 3 we see this architecture in action for simulating a specific battlefield scenario. Section 4 contains the implementation details of the architecture and battlefield scenario. In section 5 we talk about other approaches to agent coordination. Finally in section 6 we have the conclusion and future research directions.

2. Agent Coordination Architecture

2.1 Some Concepts

Actual reality is an abstraction of the world as it really exists. This includes information about each agent that exists, its position, status, speed, etc. as well as physical characteristics of the environment.

Perceived reality is specific to each individual agent and may or may not be the same as actual reality. Agents take action based upon their perception of reality

2.2 Proposed Architecture

Figure 1 depicts the overall architecture for agent coordination. In this architecture we have four major parts: Environment, EJB coordinator, web service and agents.

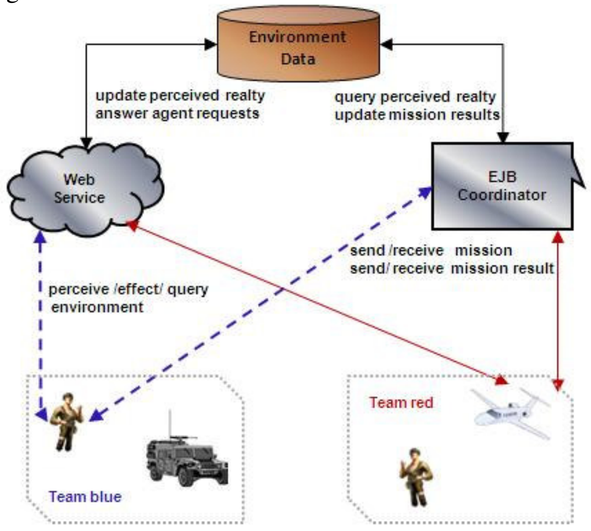

Figure 1. Overall view of architecture

Agents are objects which have independent existence and run in their own thread. They are mission oriented in that they ask for a mission from their superior, and then attempt to carry it out. The role of the superior is played by the EJB coordinator. Agents contact the EJB coordinator to get their mission initially or inform it about the result of their previous mission. While they are busy with their current mission, they use web service methods to be informed about the environment and cause changes to their environment. Agents do not communicate among themselves, but ultimately report their results and information to the coordinator.

The EJB coordinator has information about the environment, and serves agents by giving them their initial mission. The coordinator updates its perception of the world by benefitting from the perceptions of agents which are communicating to it. In this sense, agents serve the coordinator, since they share their perception of the world with it.

Environment is the world in which the agents operate. Agents as well as the coordinator usually have an imperfect view of the environment since their view is based on perceptions.

Web service is the eyes, ears and affecters of the agents. Web service has full view of the reality, but reveals only part of it to agents when they ask for information about the environment. This simulates the real world in that individuals rarely have a prefect view of their environment. The web service also makes changes to the environment that are the result of the action of agents.

3. Demonstration of proposed architecture

The architecture described above is used to simulate a battlefield environment in which two hostile teams of agents try to defeat each other.

Defeating the opposite team by killing their forces and conquering their lands is the large scale goal of each team, which is composed of smaller goals assigned to agents as missions.

For this simulation, there is only one coordinator which is used for both sides. It distinguishes agents according to the team they belong to and generates different goals for them. The web service behaves similarly and works for both teams. Hence both the web service and the coordinator are parameterized by the type of the teams. This saves code duplication.

Agents in this simulation are military units able to move across the map, approach to target, explore the battlefield, detect hidden enemies and attack them, occupy land, check for possible threats to them, retreat, reload ammunition, repair and resign of the simulation.

3.1 Scenario

The environment is defined as a square map of size 400 * 400 Km^2 in which units of two teams, red and blue, act against each other.

Initially, as soon as an agent is launched to the environment, it gets its first mission from the EJB-coordinator and starts its goal based behavior. If its mission is not to wait or resign, it starts to act to accomplishing its mission. While the agent is alive it can move in the environment and get the latest information about goals, check possible threats to it and react to them, attempt to reach the goal, effect other agents in the environment and report results to the coordinator.

The simulation can run for a specified period of time, or until one team defeats the other one, or the coordinator can not make new missions for agents. While the simulation runs, agents log their activities and coordinator saves the result of missions returned by agent in the data base. The types of units available in this simulation are airplane, tank, soldier and scout. Physical structures such as buildings, bridges, etc. are treated as special kind of agents which have no movement.

3.2 Mission Assignment

The coordinator considers the requester type to make a suitable mission for it. For example, a tank is not able to attack a plane. At mission assignment time, the coordinator checks the perceived reality to find alive enemy units and generate a mission for the requester. If there is no enemy listed after querying the perceived reality or if there is no enemy compatible with the unit type capabilities, the returned mission can be "RESIGN" or "WAIT".

If the coordinator returns "RESIGN" as the mission, the requester unit returns to base, stops and becomes inactive. In this state it can be a target but it can not be a danger to others. It still exists in the actual reality and maybe in an enemy's perceived reality. If the coordinator returns "WAIT" as the mission, for example in the case when there is no enemy which can be engaged by the requesting unit, the requester unit waits for a period of time asks again for a mission at a later time. During this waiting time, some enemy units may be revealed by team mates and added to perceived reality, and can be assigned as a target to be engaged in by this unit.

Other possible missions for a unit are "BOMB", "ATTACK", "OCCUPY" or "RETREAT" which are assigned based on the mission requester unit type and selected enemy type. For example, if the requester type is plane, and enemy type is one of ground units, the mission will be "BOMB", but if both requester unit and enemy unit are planes, the mission can be "ATTACK".

When a unit receives a mission, it starts moving toward the target position to accomplish the mission. Units use different methods to approach the enemy: they may go directly to the exact position of the enemy (such as a plane on a mission to bomb a building), approach until the enemy is in range (such as a scout on a sharpshooting mission), or keep approaching the enemy event when the enemy is within range (such as a foot soldier on a conquer mission) .

If an agent's ammunition or health is less than a minimum threshold, the mission for the unit is "RETREAT": it returns to base, reloads ammunition, obtains necessary repairs to increase its wellbeing and asks for another mission.

3.3 Some Unit Actions While Operating

Move: Unit starts moving toward the exact position of the target until it is in the right position and then it takes next step required to finish the mission.

Follow: Unit starts chasing the enemy, shoots it if it is in range and continues chasing if it became out of range. This continues until one of them is killed, or escapes because of eminent danger, or the unit runs out of ammunition.

Approach: Unit approaches enemy till it comes in range and then it shoots. This movement style is used only for scout units.

Sharp Shooting: When an enemy is in range, a scout units shoots the enemy soldier and causes 100% damage to the enemy.

Shooting: When an enemy is range, the unit shoots the enemy to destroy it. The damage caused to enemy is a function of the distance between the unit and its target. The closer the enemy, the more accurate the shooting.

Bombing: A plane drops a bomb when it is over the target and this causes a random amount of damage to the target.

Observation: Units need to observe their surroundings to find out if there is any potential threat to them. If there is an eminent danger from enemy units, the unit may decide to engage the enemy or escape, based on the number of enemies and their type. In either of these cases, the primary mission is suspended and the unit switches to the new temporary mission. If it survives the immediate threat, it retrieves the original mission and moves to achieve it.

3.4 Path Finding and Modification

It is important for units to know the latest location of their non-static target while they are moving toward it. This helps them to avoid wasting time by going to a location that the target has already left. For this purpose, units periodically check the new location of their target using the web service and modify their direction if necessary.

Agents find their direction toward their target using global coordinates. Intersection of the x axis and the agent speed vector towards the enemy forms angle α at any given time, which shows the direction towards which the agent should go. An agent based on its speed v can go a distance $d= v.t$.

Agents make the calculations shown in figure 2 to modify their path based on α and their speed.

(xt, yt)	*Target position*
(xs, ys)	*Current position of agent*
(xs', ys')	*New position of agent*
$D = \sqrt{(xt - xs)^2 + (yt - ys)^2}$	*Distance to enemy*
$\cos\alpha = \frac{xt - xs}{D}$	
$\sin\alpha = \frac{yt - ys}{D}$	
$d = v.dt$	*Distance gone by agent in a unit of time*
$dx = d.\cos\alpha$	*X component of distance per unit time*
$dy = d.\sin\alpha$	*Y component of distance per unit time*
$xs' = xs + dx$	*Computing* new *position of agent*
$ys' = ys + dy$	

Figure 2. Calculation necessary to find the direction toward target

The computation depicted in figure 2 will be repeated many times by each agent on its way to its target. Their position in the environment is updated by a call to a web service method. The target position will be observed before each new calculation, since it may have changed from the last observation.

3.5 Mission Results

A separate log is maintained for each agent. If the agent is "killed" during the operation, it aborts its last action, stops and changes its state to "KILLED". Whether the unit survives or not, the mission results are sent to the coordinator which logs the results. A mission result for an agent can be one of "FAILED", "ACCOMPLISHED", or "ACOMPALISHED BY TEAMMATES".

3.6 Scaling, Timing and Logging

In this simulation, one second is scaled to five minutes in the real world and distance in kilometers. During the simulation, each unit logs its activities as time passes. When a unit takes action at a certain time it sends the event time and explanation about the action to the logger, hence it assures that actual time of the event is logged.

3.7 Running the Simulation

This simulation can run for a specified duration of time, or as long as one side defeats the other side. When time is over, all units abort their operation, report their mission result, save their logs and then stop.

The first step to start the simulation is to create units of each side with preferred types. During the simulation, no new unit can be created but prebuilt units can be removed if they are killed or resigned.

When a unit is created, it inserts itself to the actual reality by calling a web service method. Units of one team are launched on the left half of the map and units of the other team two are launched on the right side of the map in random positions. When they are added to the actual reality, they have to be activated. Static objects (units) like lands and buildings do not need to be activated. If non-static units are created but not activated, they can become targets for the enemy, and they can not defend themselves. They are considered to be inactive, but not dead.

4 Implementation

4.1 Web Service

A web service can be deployed to an EJB container, or a web container. If it is deployed to an EJB container, it is accessible only to clients of that EJB. In order to achieve maximum scalability and platform independence and be accessible by all agents, regardless of the agents' location, we deployed our web service to a web container. Our web service is bundled with a class which contains a data source, connection pool, and necessary code to run queries on the data base. The database contains information about the environment. This information is exposed by web service methods to agents, allowing them to sense and effect the environment.

Our web service has been implemented using JAX-WS [5], which is a technology for building web services and clients that communicate using XML. In JAX-WS, a web service operation invocation is represented by an XML-based protocol such as SOAP. JAX-WS runtime system converts the API calls and responses to and from SOAP[5][7] messages. Part of the web service WSDL file is depicted in figure 3.

```
<operation name="updateUnitPosition">
<input message="tns:updateUnitPosition"></input>
</operation>
<operation name="searchUnitByMid">
<input message="tns:searchUnitByMid"></input>
<output
message="tns:searchUnitByMidResponse"></output>
</operation>
```

Figure 3. A portion of implemented web service WSDL file

4.2 Agent Implementation

All types of agents in this simulation are instances of classes that are subclasses of a common ancestor class “Agent” or the “Agent” class itself. The Agent class implements the “runnable” interface of java and defines the basic behavior of all agents. It also contains an internal thread which controls the life cycle and activities of the agent.

```
TANK TANK1 = new TANK("Red_TANK1",team, visible);
TANK1.Go(t);
```

Figure 4. Instantiating and activating a tank in simulation

Agents of both sides exist and run in one client application. This client application can be deployed to a different VJM or machine. In our case, it runs on the same system as the EJB container. Different teams of agents can run in different client applications and maybe on different machines as well.

Agents are launched to the simulation environment by calling their constructor with necessary parameters. Figure 4 depicts the creation of a tank unit. Inside the constructor, the “insertAgent()” web service method is called to insert information about the agent into the actual reality database. Agents won’t be functional and dynamic until their life cycle is started. Each agent instance has a “run” method which is executed in a separate “Thread” object. This Thread object is the value of an instance variable called “runner” of the “Agent” class. Calling the “Go()” method of an agent starts its thread and independent existence. Figure 5 depicts the “Go()” method of the “Agent” class.

```
public void Go(int t)
{
 Agent.finishTime = t;
 addLog(System.currentTimeMillis(), "Unit started");
 addLog(System.currentTimeMillis(), "Total simulation time
is set to " + Agent.finishTime);
 this.agentNum++;
 this.runner.setName(this.getMid());
 this.runner.start();
}
```

Figure 5. “Go()” method of the “Agent” class

The “run()” method of the “Agent” class is depicted in figure 6. All activities of the agent take place in the while loop of the “run()” method.

The only way to get out of the loop is to be killed or resign from the simulation, in which case the instance variable “runner” is set to null and the loop terminates. When the agent is killed or resigns, it saves the agent operation log.

```
public void run()
 {
   addLog(System.currentTimeMillis(),"Unit is
      running");
   Thread thisThread = Thread.currentThread();

   while (this.runner == thisThread) {
     askMission();
     doMission();
    }

    saveLog(this.log);
  }
```

Figure 6. Agent’s life and action control in run method of the thread.

4.3 EJB Coordinator

The coordinator is a set of EJBs, one of which communicates with the agent and uses other EJBs to perform its duties. The coordinator serves each single agent in the environment. Each team of agents could have their own EJB coordinator, but in our implementation, one coordinator serves both teams. Inside the EJB container, beans can serve each other internally by a remote or local interface. We have used the remote interface of EJBs to enable our Entity Beans making up the coordinator to be distributed on different machines if required.

The coordinator contains two entity beans: “UnitEntity” and “MissionEntity” representing agent and goal entities. These entity beans are used to represent agents and missions of the simulation as objects in the database that keeps the environment data. The entity manager and persistence unit of Java EE is used to manipulate the environment database. The EntityManager API creates and removes persistent entity instances, finds entities by the entity’s primary key, and allows queries to be run on entities [5] [6].

Two session beans are developed to implement the logic of coordination. Clients use the remote interface of these session beans for getting their tasks.

Agents, before they can call methods of the remote interface, have to find a reference to the remote interface by performing a JNDI lookup [6]. Figure gives a better view of the concept.

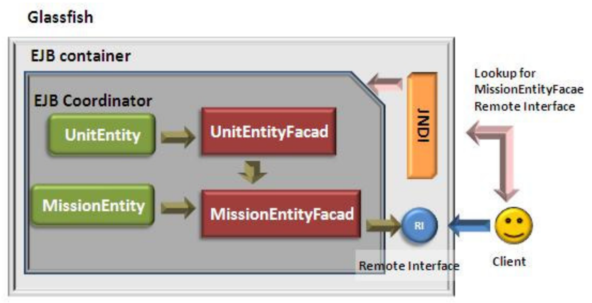

Figure 7. Components of coordinator

5. Related Work

We have not found any research that uses the same set of tools and techniques as we do for agent coordination. There is a lot of research in the area of agent coordination. We give a small, representative survey below.

In [7], the authors investigate the degree to which intelligent agent coordination strategies scale along various dimensions of stress.

In [8], the authors describe the use of coherence constraints as a means to regulate agent interaction. Coherence constraints describe relationships between the content of utterances, and the context.

In [9], the authors describe a novel approach for using centralized "single-agent" policies in decentralized multi-agent systems by maintaining and reasoning over the possible joint beliefs of the team. This approach offers strategies to reducing the amount of communication of the multiagent coordination system.

A channel-based exogenous coordination language, called Reo, and discuss its application to multi-agent systems is described in [10]. Reo supports a specific notion of compositionality for multi-agent systems that enables the composition and coordination of both individual agents as well as multi-agent systems.

6. Conclusion and Future Work

In this paper we described an agent coordination architecture based on XML web services, Enterprise Java Beans and light weight java threads.

Our architecture is both extendable and scalable since agents can exist in different java virtual machines or even on different physical machines and still live in the same virtual environment. This is made possible through XML web services and the EJB technology employed. We demonstrated the utility of our architecture in simulating a battlefield scenario consisting of two opposing teams where each team contains different kinds of agents representing combat elements, such as tanks, soldiers and planes.

For future work, we are planning to split the coordination job among EJBs in a hierarchical manner, whereby EJBs communicate with one another using a messaging service such as JMS . This will allow more complex coordination jobs to be handled. We are also planning to have a visual representation of agents, so that their activities can be seen in real time.

References

1. H S Nwana, L Lee and N R Jennings (1996): "Co-ordination in software agent systems.", BT Technol J, Vol 14, No 4

2. Weiming Shen; Hamada Ghenniwa; Yinsheng Li (2006): "Agent-Based Service-Oriented Computing and Applications", *Pervasive Computing and Applications, 2006 1st International Symposium,* 2006, pp. 8 – 9.

3. Zhi-Zhong Sun; Bin Li; Liang Li (2007):"An Adaptive Agent Coordination Framework for Web Services Composition", *Machine Learning and Cybernetics, 2007 International Conference,* Volume 7, 19-22, pp. 3870 - 3875

4. R. Scott Cost, Yannis K Labrou,Tim Finin (2000): "Agent Communication Languages and Agent Coordination", *Coordination of Internet Agents: Models, Technologies and Applications.* July 01, 2000. Springer-Verlag

5. The Java™ EE 5 Tutorial Third Edition http://www.sun.com, Accessed 2 Jun 2008

6. Rima Patel Sriganesh;Gerald Brose;Micah Silverman (2006): "Mastering Enterprise JavaBeans 3". Wiley Publishing Inc. ISBN 978-0-471-78541-5

7. Durfee, Edmund H. (2001): "*Scaling Up Agent Coordination Strategies*", Computer, Vol.34 Issue 7, pp.39-46

8. Joris Hulstijn , Frank Dignum, Mehdi Dastani ,(2005): "*Coherence Constraints for Agent Interaction*", LNCS 3396/2005, Springer Berlin / Heidelberg Publishing, ISBN 978-3-540-25015-9, pp. 134-152

9. Maayan Roth, Reid Simmons, Manuela Veloso ,(2005): "*Decentralized communication strategies for coordinated Multi_Agents Policies*", Multi-Robot Systems. From Swarms to Intelligent Automata Volume III, Springer Netherlands,ISBN 978-1-4020-3388-9, pp. 93-105

10. Dastani, M., Arbab, F., and de Boer, F. (2005): "*Coordination and composition in multi-agent systems.* " Proceedings of the Fourth international Joint Conference on Autonomous Agents and Multiagent Systems (The Netherlands, July 25 - 29, 2005). AAMAS '05. ACM, New York, NY, 439-446