Janet: A Lightweigt Distributed Multi-Agent System in Java with Automatic Agent Load Balancing

Oliver Plohmann, Alois Schütte

University of Applied Sciences, Department of Computer Science,
D-64295 Darmstadt, Germany
e-mail: oliver@plohmann.com, a.schuette@fbi.fh-darmstadt.de

Abstract

This paper briefly describes design and implementation of Janet, which is a distributed multi-agent system written in Java and XML. Janet provides are platform for writing agent-based distributed applications. The basic layer called cooperative agent system (Janet.CAS) provides are platform for writing agent-based distributed applications. Users define applications consisting of agents with capabilities. Applications dispose of their own object space for their agents to live in. Applications are protected from each other and may be distributed over several machines. Agent load balancing is implemented in the layer for automatic distributed execution (Janet.ADÉ) as a cooperative agent-oriented system based on Janet.CAS. Agents can be suspended and migrated to machines with lower CPU usage where they resume execution. The suspension process is semi-transparent since the developer has to assist the system in suspending an agent selected for eviction and in storing its execution context. While Janet in its current state cannot compete with more elaborated agent systems such as Jade [BPR99] it manages to offer an interesting combination of a distributed multi-agent platform, an application-level workload balancing system, and a distributed programming system based on the agent paradigm with distributed asynchronous execution with or without load balancing. Janet was developed in partial fulfillment of a Master's degree in Computer Science. This document summarizes the thesis [OPL04] handed in at the University of Applied Sciences, Department of Computer Science, Darmstadt, Germany.

Introduction

The idea of this thesis work is to develop a system for automatic load balancing. The concept for the system is developed using a *multi-agent paradigm*, where distributed agents cooperatively achieve a common goal.

Distributed operating systems like Amoeba [TAN01] or Sprite [OUS88] offer load balancing on operating system-level. On the contrary, the system presented in this thesis is implemented on *application-level*. There are several advantages to application-level load balancing: It is possible for a Java application to make use of workload balancing even if it was not designed for workload balancing to begin with. There is also no need to migrate to a special distributed operating system for which

often little software exists and that are even discontinued after they are no longer needed as a research vehicle. Load distribution has to manage the mapping of distribution objects to distribution units [SCH95]. Distribution objects may correspond, for example, to processes, threads, or data structures whereas distribution units may correspond, for example, to computing nodes or application processes. For load balancing on application-level the mapping of distribution objects to distribution units can be adjusted to the implementation of distributed objects. For distributed operating systems this mapping has to be made automatically, which results in a loss of flexibility and additional overhead. An advantage of distributed operating systems is that they provide workload balancing in general for all applications and not only for individual ones. However, it is believed that there is a place for a system like Janet in areas where distributed operating systems or cluster systems are too effortful or expensive. In addition, Janet is not only a system for workload balancing in Java but also a general-purpose multi-agent platform.

Simplifications

For being able to implement Janet in the time available for writing the thesis several simplifications had to be made.

For the time being Janet does not support an explicit ontology or ACL. Ontologies and ACLs still seem to be in a state of ongoing research and development. It therefore appears not appropriate to make a choice in favor of a specific ontology or ACL at the time of writing. This is contrary to research-oriented agent systems where using ontologies and ACLs is part of the research work. The approach in Janet is to wait and see how ontologies and ACLs will develop in the future. However, since a user of Janet has full control of her application, she can plug in whatever ontology and ACL she wants to use.

Security concerns are ignored for the time being. Respective security mechanisms can be added in a later stage.

Current CPU load of the hosting machine is not read in for the initial implementation. CPU load and load changes are simulated by a user-changeable threshold value.

Design of Janet.CAS

Commands and Interpreters

Communication between agents is based on the distributed asynchronous exchange of command objects with agents running in their own threads. When an agent receives a command it invokes an interpreter it chose of its own to handle the command. The command-interpreter pair in Janet.CAS has been developed as an extension of the command pattern as described in [GA95]. The split between command object and interpreter object is necessary to ensure the autonomy of an agent. If an agent were simply told by a command, sent by some other agent, what to do, it would only be an object receiving a message rather than an autonomous agent receiving an event and reacting to it in a way it assumes appropriate.

Applications and Capabilities

The application object in Janet.CAS allows the user to register her application with Janet so that the agents defined by these applications will be instantiated and started. A node object serves as a host for applications on one machine. However, a machine may host several nodes. A node itself is an application with capabilities that have system privilege. An application is identified by its name. Applications with the same name on different nodes are considered by Janet to be the same application. An application's definition does not need to be identical on every node, though. Only agents of the same application can communicate *directly* with each other to protect agents of an application to be infringed by agents of a different application. Agents of different applications can only communicate indirectly with each other through events. A capability consists of a set of interpreters and is associated with one ore more agents. Capabilities are a means to partition an application into logical sub-applications. It is thinkable to assign agents to more than one capability. This idea has been dropped since the algorithm for load balancing would become much more difficult.

Schedulers

A scheduler is used to execute interpreters, associated with commands, on behalf of the agent. Implementation-wise the scheduler is an active object, which means that it runs in its own thread, whereas the agent object merely serves as an anchor object to accept commands and to expose its public interface to other agents. When an agent receives a command from another agent it places the command in its scheduler's command queue. Commands are executed one after the other. The scheduler looks up the interpreter associated with the command, passes the command over to the interpreter and starts it. The interpreter extracts any required input parameters from the command. If not suspended for eviction an interpreter runs to completion. Several agents may share the same scheduler, which improves performance considerably in case several thousand agents exist that are running concurrently as shown in [BRHL99]. A node is maintained by one or many system agents of which the schedulers run at higher priority than the schedulers of the user agents in order to preserve a node's liveliness.

Object Spaces and Event Registries

Object spaces are associative object stores where agents' interpreters can store permanent information. There are object spaces on application-level, node-level, and cluster level. Agents that belong to the same application on the same node may store objects in the application-level object space. Agents of all applications on the same node may store objects in the node-level object space. All agents of all applications of all nodes may store objects in the cluster-level object space. There can be several cluster-level object spaces, which are implemented as RMI server objects. For a cluster-level object space to be visible for a node it has to be defined in the node's XML descriptor.

Event registries allow agents to register for events that can be signalled by any other agent, residing on whatever node and pertaining to whatever application. Analogously to object spaces, there are event registries on three levels: application-level, node-

level, and cluster-level. The cluster-level event registry is implemented as an RMI server object. For a cluster-level event registry to be visible for a node it has to be defined in the node's XML descriptor.

Sending Commands to Agents

Being true to the agent paradigm only an agent can send a command to another agent. Since the executable part of an agent is comprised of the interpreters of its capability it is only possible to send a command to another agent from within the execute method of an interpreter.

```
public class MyInterpreter implements IInterpreter
{
    // …
        public void execute(CommandAccessor cmdAccessor)
        {
                if(cmdAccessor.getCommand() instanceof MyCommand)
                {
                        ICommand myCommand = new MyCommand();
                        IAgentProxy agent = cmdAccessor.getAgent("nodeName");
                        agent.accept(myCommand);
                 }
         }
    // …
}
```

Figure 1: Sample interpreter used by an agent to send a command to an agent with the same name of the same application and capability residing on node "nodeName"

As shown in figure 1 the interpreter retrieves a proxy to the destination agent and sends a command to it. The proxy takes care of passing the command to the destination agent and hides the complexity of getting this task accomplished. Figure 1 a very simple case where the destination agent has the same name as the sending agent and resides on a node named "nodeName". There is no way to obtain a proxy to an agent belonging to a different application than the sending agent.

In figure 2 on the next page a more "sophisticated" example is shown where the complete path to the destination agent is specified to obtain a proxy to it.

```
// …

if(cmdAccessor.getCommand() instanceof MyCommand)
{
        AgentPath agentPath = new AgentPath("nodeName", "myCapability",
        "myAgentName");
        IAgentProxy agent = cmdAccessor.getAgent(agentPath);
        ICommand myCommand = new MyCommand();
        agent.accept(myCommand);
}

// …
```

Figure 2: Sending a command to an agent specifying an AgentPath.

Node Registry

The node registry knows about all applications on a node with theirs capabilities and their agents. Every node in a group of nodes has a local copy of the other nodes' registry. This makes sure that agent look-up is immediate, which is important in order to react swiftly to load imbalances.

Every node having a complete image of all other node registries in a cluster may result in a scalability problem since synchronizing node registries requires more command sends between nodes' system agents the more nodes exist in the cluster. This is a problem in the current implementation of Janet, which is especially an issue when starting up or shutting down nodes. The envisioned solution is only to notify one or a few master nodes immediately after a node registry change and to synchronize the remaining node registries in the background. When an agent wants to deregister it has to wait for the acknowledgement of all nodes till it is allowed to physically remove itself. Deregistering an agent therefore takes some time. The recommended approach is to have agents with a long lifetime.

Design of Janet.ADÉ

Suspendable Interpreters

Load balancing in Janet is realized by suspending executing interpreters and evicting them to nodes with a matching capability. When defining an interpreter the user has the choice to implement either the regular interpreter interface provided by Janet.CAS or the derived interface for suspendable interpreters provided by Janet.ADÉ. Janet does not level out load imbalances by migrating agents. Instead it evicts commands. This has many advantages. There is no severe security problem as with mobile agents. Evicting entire agents is also costly in many ways. Firstly, the agent has to be transferred with all its commands and permanent data. Secondly, it has to be deregistered from the origin node and registered at the destination node. This is a time consuming process which makes immediate response to load changes impossible. In addition, when evicting individual commands instead of entire agents, load can be balanced at a finer level of granularity.

Executor-Observer-Distributor Triad

Three different kind of nodes are defined to accomplish load balancing. The different roles of these nodes are defined by adding an extra capability to their system application. An *executor node* carries user defined applications. If the user wants her application to benefit from agent load balancing, she needs to add suspendable interpreters to her application's capabilities. Then her interpreters are visible to the load balancing system and will be considered for eviction.

Every workstation in a cluster needs to have a single *observer node*. The sole purpose of an observer is to observe the workstation's CPU load. If a certain CPU load threshold value is exceeded or fallen below the distributor node is notified. In the former case the distributor will make all commands of all executor nodes on the workstation be evicted to other executor nodes with matching capabilities on other workstations (full eviction). In the later case all executor nodes on the workstation the observer resides on are re-considered for hosting evicted commands from other workstation's executor nodes.

An executor informs the distributor node whenever its load changes. When the distributor sees that an executor is about to change to the idle state it evicts a suspendable interpreter from a different executor on another node with executing or waiting suspendable interpreters of matching capability (partial eviction). This approach is called recipient-initiated eviction (the opposite is called sender-initiated eviction). The executor receives a command from the distributor to evict an interpreter of a specific capability. The executor's interpreter that suspends the user's interpreter is added to the executor's core capability to make sure it is executed with highest priority, thus being able to interrupt the user's interpreter in order to suspend and evict it.

Load Determination

There are several algorithms to detect load in a cluster. Several of these algorithms are mentioned in the thesis. One way to quantify a system's load is to determine the length of the list of running processes. This approach is called ShortestQueue. Since commands in Janet are placed in queues and then processed sequentially it is easy to determine the overall queue length of an executor. This makes applying the ShortestQueue algorithm in Janet straight-forward, which is one reason this algorithm was chosen. Another reason is that it delivers good results and does not require additional information from the Java virtual machine that is difficult to obtain without changing the virtual machine or host operating system.

Queue Size Categories and Capability Queue Sizes

As mentioned earlier in the text an executor notifies the distributor whenever its load changes. Notifying the distributor whenever any of the executor's queue changes in length would result in considerable command traffic in the cluster. To reduce command traffic Queue Size Categories (QSCs) are introduced, which allow classification of queue lengths. The distributor only receives a load change notification from the executor when the executor changes QSC. The user may define several QSCs for her executor node. A QSC definition must always have a QSC

named "0" (QSC0), which is considered by the system to represent the idle state, and a QSC named "1" (QSC1), which is considered the state by the system where the executor is just about to become idle. Additional QSCs are optional. It is recommended to define more QSCs than only QSC0 and QSC1. QSCs can be defined individually per node and workstation. By defining the same QSCs for different executors or workstations individually the user has a means to reflect different performance of different hardware.

Applications and capabilities are convenient for partitioning an agent's executable part. However, these elements make finding a decision how to react to a load change more effortful. To find an executor with least load to accept an evicted command the distributor needs to consider all nodes' QSCs and it needs to make sure that candidate executor nodes dispose of the required capabilities. To address the later issue the concept of Capability Queue Size (CQS) has been introduced. The CQS of an executor is the sum of the number of all executing or waiting commands of all agents on an executor of a specific capability. Whenever an executor changes QSC it attaches a list with the current CQS values of all its capabilities to the notification command sent to the distributor. When looking for the least loaded suitable executor a distributor combines QSCs and CQSs of the cluster's executors to make an eviction decision. For further details the interested user is referred to the thesis [OPL04] document.

Open Issues

There are several open issued that need to be closed till the first version of the system can leave the beta state (besides doing some more testing). The most important issues are listed below:

- Agent aliasing: In the current state of the system a remote agent has to be narrowed by specifying its complete physical location consisting of node loaction, application name, capability name, and agent name. Alternatively, an agent needs to be addressable through an alias.

- There are scalability issues that need to be dealt with, because in Janet every node has full information in it registry about all the applications with their capabilities and agents that exist on other node. This issue has already been mentioned earlier in the document.

- Observers are currently running in simulated mode. They do not retrieve the workstation's CPU load from the host operating system somehow. Using simulated observers is very effective when testing the system since it is easy to create any kind of load change scenarios. Nevertheless, an interpreter needs to be added to the observer that reads out the CPU load from the host operating system.

- At the moment interpreters once their execution has been started run to completion. An agent can therefore not respond to an external event until the currently executing interpreter has finished execution. There are several approaches possible to address this issue. For example, Jade [BRHL99] applies "interleaving of conversations".

- The communication system has no pluggable interface. Exchanging RMI for command distribution with some other middleware is not straight-forward at the moment.

Future Directions

The general idea of Janet is to provide a system for distributed processing. The agent-paradigm is applied, because it offers an interesting approach for doing distributed programming. However, this work does not attempt to prove the usefulness of agents as such. Agents are being made use of as long as they proof to be useful. Concepts such as ACLs and ontology that in over ten years time have not managed to gain momentum are not further pushed. Following this spirit, the general direction for future work on Janet is to add functionality that makes it more useful for distributed programming while the idea of asynchronous and autonomous processing as embodied by agents remains a core design decision.

The future plan for Janet is to publish it and look for developers that would like to join in to add new functionality or improve conceptual issues. A true clustering solution that supports failover could be implemented as a research project. Security has to be added so that only commands are accepted from trusted sources and can only be sent to trusted destinations on a secure connection. Functionality could be added to make agents migrate as well. This would make sense to react to imbalances in memory usage rather than to react to imbalances in CPU consumption.

Summary

A platform for distributed cooperative agents in Java has been developed. The platform offers distributed event notification and distributed object spaces. It provides an interface for users to plug in their custom applications that make use of the agent platform. Instead of invoking methods on agents, an alternative paradigm is used where agents communicate through the asynchronous exchange of commands to which they respond with the execution of an interpreter they have chosen autonomously.

The workload distribution layer is completely separated from the agent platform. It allows to level out load imbalances by evicting commands from overloaded agents to lightly loaded agents on other nodes. Since interpreters are inserted into scheduler queues, detecting lightly loaded agents using the ShortestQueue algorithm is straight-forward. The user has to assist the balancing system by implementing commands eligible for workload balancing in such a way that the system can suspend them before migration. With this slight sacrifice in transparency workload balancing could be implemented for Java without having to modify the Java virtual machine to provide transparent thread migration.

Summing up, the concept used by the developed system represents an interesting compromise between workload distribution systems on system-level that cannot be tailored to an application’s specific requirements and systems that leave load distribution completely to the programmer. The system offers an interesting combination of an agent platform and a workload balancing system.

Janet is public and may be freely used. Documentation, sources and binaries are available at:

http://www.objectscape.org/janet.

References

[BRHL99] P. Busetta , R. Rönnquist, A. Hodgson, A. Lucas, “Light-Weight Intelligent Software in Simulation”, Agent Oriented Software Pty. Ltd., Melbourne, Australia, 1999.

[BPR99] F.Bellifemine, A.Poggi, G.Rimassa, “JADE – A FIPA-compliant agent framework”. In: Proceedings of the Practical Applications of Intelligent Agents, April 1999 (see http://jmvidal.cse.sc.edu/library/jade.pdf).

[GA95] E.Gamma, R.Helm, R.Johnson, J.Vlissides, “Design Patterns”, Addison-Wesley, Reading, Massachusetts, 1995.

[OPL04] O.Plohmann, “A System for Automatic Distributed Execution in Java using Multi-Agents - Concept Development and Implementation”, Master Thesis, University of Applied Sciences, Department of Computer Science, Darmstadt, Germany, 2004.

[OUS88] J.K. Ousterhout et al.: “The Sprite Network Operating System”, Computer Magazine of the Computer Group News of the IEEE Computer Group Society, ACM CR 8905-0314, 1988.

[SCH95] T. Schnekenburger, “The ALDY Load Distribution System”, Institute of Computer Science, Munich Technical University, 1995.

[TAN01] A. Tanenbaum, “Modern Operating Systems”, Prentice Hall 2001.