2002

Net.Sense

John Mikucki

Rochester Institute of Technology
Department of Computer Science
Thesis

Net.Sense

By

John J. Mikucki

A thesis submitted to
The Faculty of the Computer Science Department
In partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Approved by:

Dr. Julie A. Adams
Chairperson

Dr. James R. Vallino
Observer

Mr. Paul T. Tymann
Reader

December 2, 2002

Rochester Institute of Technology
Department of Computer Science
Thesis

Net.Sense

By

John J. Mikucki

32 King St
Danbury, CT 06811
Phone: Unlisted

Date: December 3, 2002

_____

John J Mikucki

Contents

List of Figures

Part I

Background

Missing Page

Chapter 1

Introduction

As computation becomes increasingly integral to our daily lives, we build ever larger networks of ever more complicated computing devices. Techniques for managing this complexity lag behind our networks' growth. System administrators spend their time 'fighting fires' rather than addressing the bigger issues. Net.Sense will serve as a proof-of-concept of a new type of network management system, using biological models and statistical principles to address scalability, predictability, and reliability issues associated with managing the highly complex computer systems that we as a society have come to depend on.

1.0.1 Function

Net.Sense should allow system administrators to set policies for managed networks and thereafter let the system manage itself. The administrator is able to monitor the aggregate network and system state, examine individual machines, and obtain prompt warnings about out-of-tolerance state and recommended actions. Net.Sense can be granted global or conditional autonomy as regards taking action on the managed network.

To use Net.Sense, the system administrator specifies optimal operating conditions for the managed network. He or she may create additional sensor/effector packages to measure and control additional pieces of the managed systems' state. Finally, the system administrator crafts rules and policies that describe how new effectors change the system state, and the conditions upon the effectors' use. Net.Sense then assumes primary responsibilty for reaching and maintaining optimal operating conditions within the managed network.

1.1 Issues Addressed

As mentioned above, Net.Sense is intended to address three concerns with existing system administration tools: scalability, predictability, and reliability. Consider the problem of log collection over a network. The system administrator

responsible for managing the network schedules a cron job to run at, perhaps, five minutes past every hour. Logs are collected, processed centrally, and all is well. However, as the network grows, so do the bandwidth requirements and log processing time. The network grows sluggish early in the hour as hundreds or thousands of machines simultaneously try to send log data to one or two servers.

The wily system administrator (sysadmin) easily sees that the solution is to distribute the load across the available time, and changes the cron jobs so that some machines transmit at 5 past the hour, some at 10 past, and so on. However, (s)he must now maintain all these separate configurations. If more machines are purchased, they must be manually assigned to groups. The process is needlessly labor-intensive.

Equally as bad, our intrepid sysadmin has been (and remains) vulnerable to the night-watchman problem. By studying network latency, a motivated attacker can determine traffic patterns, isolate the log traffic, and work around those times, beginning his or her run just after the logs are sent, the attacker has nearly a full hour to work and knows it.

If the problem is predictability, the solution is pseudorandomness. By collecting logs at random intervals, the attacker's window of opportunity becomes unpredictable, and thus more difficult to use. Careful application of statistics enables creation of useful statements about the *overall* system behavior, while continuing to frustrate would-be attackers.

Finally, Net.Sense addresses issues of scalability and availability of critical network services through distribution and replication. In a traditional network configuration, one or two failures can incapacitate a critical service. Net.Sense's highly distributed mobile architecture ensures that if a host or even a network becomes unavailable, system performance degrades gracefully rather than cataclysmically.

1.2 The Net.Sense Vision

1.2.1 Organization

The canonical Net.Sense system is composed of three agent communities: the 'swarm', the 'reasoners', and the 'actors'. The swarm community serves to collect data by randomly selecting, visiting, and inspecting hosts from the managed network. It is sometimes referred to as the sensor community, because swarm agents' primary role is to carry and use sensor packages.

The reasoner community manages and processes the collected data. In doing so, the community may determine that changes must be made to managed hosts in order to meet the managed net's operational requirements. The reasoner community is occasionally called the 'thinker' community, as most of the Net.Sense artificial intelligence (AI) and other computational overhead will be performed by this group.

Members of the actor community are responsible for taking actions within

the managed network. The actor community is sometimes called the 'effector' community, as it is responsible for effecting changes in the network.

1.2.2 Swarm Community

The swarm community consists of a relatively large collection of simple, resource-parsimonious mobile agents that randomly roam the managed network. These agents use sensor packages to collect data from their environment (potentially including themselves) and add them to the global shared InfoSpace (a blackboard-like construct which will be discussed in full in sections 5.1.3 and 6.5), where they can be reasoned over.

The swarm is loosely organized on biological and probabilistic or statistical methods. These methods provide behavior patterns predictable at the macroscale but effectively random at the microscale level. Much like an unopened soda can, the system's macroscale properties (internal can pressure) are predictable and controllable while microscale properties (when a molecule will pass through a certain region of space, when a given host will be visited) are far less so.

Successful construction and efficient operation of the swarm community provide bountiful avenues for exploration. The list includes:

- Pure pseudo-random movement
- Pseudo-random movement with limited-range tropism (pheromone model)
- Modified subsumption model, with varying priorities
- True-space movement (Movement on a degree-4 2-D lattice or grid)
- Hyperspace movement (Movement on K_n)
- Netspace movement (Movement with costs associated to the actual network topology)

1.2.3 Reasoner Community

The reasoner community is responsible for assimilating, managing, and processing collected data to monitor the managed network. Should the system's rules deem it necessary, reasoner agents can create and dispatch members of the actor community to effect change within the managed network.

There are many different aspects to the reasoner community. Some agents may be librarians, tasked to maintain the InfoSpace by sorting, collating, and retrieving data. Other, 'thinking' agents may be responsible for reasoning over the InfoSpace, deriving conclusions, suggesting courses of action, and tasking actor agents to implement the effects necessary to meet operational goals.

Individual thinking agents perform proportionately small chunks of the requisite reasoning. This enables enhanced redundancy, load-balancing, and decreased network traffic.

The current vision sees a variety of possible designs for 'thinking' agents, but all so far are variants of three archetypes: 'processors', 'detectors', and 'commanders'. Processors are responsible for performing 'infrastructure' computations, in other words, services generic enough to serve many goals. For example, statistical services would fall into this category.

Detector agents should perform forward-chaining reasoning, understand a limited section of the operational requirement's domain (for example, CPU usage or port-scanning) and 'subscribe' to updates of the relevant portions of the InfoSpace.

Commander agents are tasked with effecting the system state changes requested by thinking agents. Determination of proper implementational steps can be performed by goal-seeking logic applied to the effects requested by thinking agents and semantic 'tags' describing the effects of specific toolkits. For example, a 'process kill' toolkit might have semantic tags stating that applying the toolkit to a process will eliminate that process' memory consumption, CPU usage, reduce the user's 'degree of presence' on the machine, take a port/socket out of service, etc. Hence, the commander agent could be given the task "kill all processes owned by userid 173".

Once tasked, commander agents reequip and retask existing actors, or create and task one or more new ones. These actor agents then go out into the managed network and affect hosts by executing their payloads. Commander agents with a particular objective can monitor the progress of agents in their command by subscribing to the portions of the InfoSpace they publish. In this way, both the managed system's state and the task's progress can be kept under close watch.

For example, consider a network threat detector. This thinking entity would take sense data (firewall logs, traffic profiles, etc) and check for flaggable conditions, such as unusual network traffic patterns. The threat analyser would then prioritize threats and propose responses. Like the human brain, the threat analyser may have multiple levels of cognitive processing, in other words, each task may be handled by groups of collaborative 'subdomain expert' agents. A proposed response profile is then published into the InfoSpace for implementation by the appropriate Commander agent(s).

1.2.4 Actor community

The actor community is composed of a mixture of short- and long-lived agents. Long-lived agents exist to provide commonly-requested services to other members of the Net.Sense system. Other, more ephemeral agents may be created, equipped, and dispatched to perform a one-time task, at which point they may be taken out of service. Actor agents are envisioned to be heavily scripted, programmed to perform a specific and fixed sequence of actions to affect a system or systems in the manner selected by a thinking or commander agent, or a human overseer.

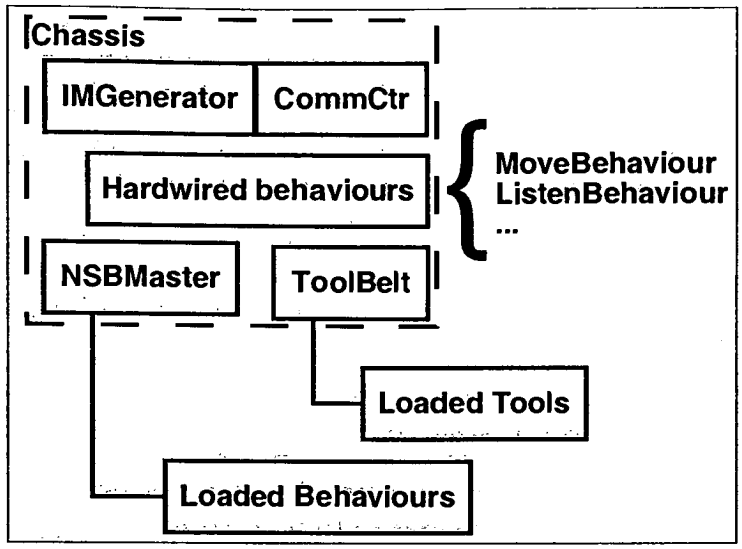


Figure 1.1: Prototype Chassis

1.3 Organizational Overview

1.3.1 Agent Design

In Net.Sense, agents are based on 'chassis'. (See Figure 1.1.) A chassis is a set of modules that are hard-coded and immutable as of compile time. Chassis support the dynamic loading (and unloading) of toolkits, and the behaviors that manage them. Agents' abilities to sense and affect arise from the toolkits with which they are equipped, and the behaviors that activate them. The collection of toolkits and controlling behaviors with which an agent is equipped is called a 'loadout'.

Some behaviors, however, provide internal or metaservices which, for various reasons, do not fit easily into the loaded behaviour system. These behaviors are hardwired into the agent chassis, and are therefore immutable and common to all agents based on that chassis. All agents will therefore have certain core capabilities (as provided by their chassis), in addition to any extended capabilities provided by their loadout.

Through their loadouts, Net.Sense agents provide services, which in turn can be used by other agents to provide their own services. User interface, reasoning, agent control, data collection are all envisioned as services provided by agent loadouts. Capable agents could therefore search for or request agents with special capabilities to assist them. For example, a forensic agent might request a network control agent to drop the network interfaces on a particular machine to preserve evidence of an intrusion.

Because the agent's 'mind' is effectively a loadable module, agent 'personal-

ities' (that is, their control and behavioral logic) are as interchangable as their capabilities. Some agents may have beliefs about the system state; others may function blindly, follow modifiable scripts, etc. For example, actor agents are typically viewed as having specific tasks (scripts) to perform. Other agents, for example, swarm agents, may use behavior modules modelled after biological models, while yet others may be highly collaborative in nature, for example service brokers, or dealers.

1.3.2 Agent Spaces

In the optimum configuration, Net.Sense agents operate in two spaces simultaneously. The first space is loosely called 'physical' space, because actions within the physical space directly affect the agents' positions, sizes, and other attributes that would be considered physical were the agents corporeal. Movement, environmental impact (bandwidth utilization, CPU utilization, memory utilization, etc) are all 'physical' concerns from the agents' perspectives.

The second space is called InfoSpace. It is the aggregate of all published medium- to long-lived knowledge held by agents in a Net.Sense system. Which information to publish and how often to update it is somewhat arbitrary. A balance must be struck between publishing each change of a loop variable and excessive system overhead. Examples of 'good' published data include agent equipment lists, locations, intentions/missions, and relevant configuration parameters. Data acquired from sensor toolkits should also be published into the InfoSpace.

1.4 Interface

The Net.Sense system must have some means of interacting with its human controllers. This is accomplished via the actor agents of the Interface family. These are ordinary agents whose loadouts provide a user interface to the InfoSpace and various commonly-used tools. The nature of the InfoSpace and its dual role as command channel and data repository makes the agent metaphor ideal as a user interface tool. The user interface (UI) agent may migrate with a user from terminal to terminal, may preserve session data, or even optimize its interface by collecting statistics on actions taken by the operator. Initial interfaces are expected to be rudimentary as they are not the focus of initial development efforts.

1.5 Functional Specification

The operational vision of the Net.Sense system described above leads to goals and ideals that are expressed as statements about the manner in which the system operates. The most important of these are listed here.

1.5.1 Agent requirements

The agent requirements describe qualities attributed to agents and their functions within the managed network.

mobility Agents must be mobile. They must be able to transport themselves in their entirety to a new location and continue operating just as before.

communication Agents must support rich communication. Communication should be transparent, robust, and automatic. Code as well as data must be transported through the network.

modularity Agents must be modular. Some type of toolkit framework is required, so that agents can be modularized and reconfigured during system operation.

robustness Agents should be robust in the face of error and reliable despite unusual or constrained operating conditions.

unobtrusive Agents should be as lightweight as possible, both in terms of static (ie memory footprint) and dynamic (ie network bandwidth) resources.

1.6 System Level Functional Requirements

The system level requirements describe the function of the system as a whole. The following three items are, generally speaking, the core of the Net.Sense vision: an introspective, self-managing process that senses its environment, reasons over it, and can in turn affect it.

Sense Requirement The system must be able to collect selected data from machines on the network. The user should be able to create new data point descriptions and collection modules, and flag them for collection as the system operates.

Affect Requirement The system must be able to take actions upon the systems it is deployed to. Some of these actions may be dangerous, privileged, or both. These actions must be capable of being protected commensurate with their associated risk level.

Reason Requirement The system must be able to assess collected data, reason over it (by pattern recognition, rule chaining, simulated neural processes, tropism models, or other AI techniques) and select a course of action that is appropriate as defined by the system administrator. Further, the administrator should have the ability to require certain actions be associated with a justification trail.

1.7 System Level Operational Requirements

The system level operational requirements describe the operational characteristics of the ideal Net.Sense system.

Weight Net.Sense must be parsimonious in its use of static (RAM, CPU) and dynamic (network bandwidth) resources. Operating a Net.Sense system should utilize less than 10% of the available network's capabilities.

Security Net.Sense provides greater utility as its autonomy and power increases. Therefore, maximum utility should be achieved when the system has complete (root) power, and is free to act 'at will'. However, operating in such a manner poses significant risk to correct system operation, should the system fail or be hijacked. Therefore, Net.Sense should provide assurances of fail-safe and secure operation. (Strong authentication, encrypted & signed toolkits, secure coding, etc.)

1.8 System Level Engineering Guidelines

The system level engineering guidelines are for system implementors, who should keep these points in mind when making any Net.Sense-related design decision.

Modularity The Net.Sense system must be engineered in such a way that it is easily expandable and configurable. It must be simple for the operator(s) to add or enhance system functionality as the system operates.

Security Many effects Net.Sense systems will wish to perform will require elevated privileges. Further, Net.Sense systems may contain and manipulate sensitive system information. The end-user system administrator must have assurances about the degree of risk (s)he assumes by running a Net.Sense system, and Net.Sense developers should take steps to minimize this risk wherever possible.

Scalability Net.Sense should scale easily from ten-host workgroups to thousand-host multinational corporate systems and, if possible, beyond. Net.Sense developers should make reasonable efforts to ensure their extensions are architected for scale.

Extensibility Net.Sense developers should try to modularize their code in such a way that it maximises the utility others will derive from it. Do not lock users into any one way of doing things save where interoperability demands it.

1.9 Principal deliverables

The principal deliverables of this thesis are:

1. Architectural design documents (see Chapter 5)
2. Prototype framework implementation (see Chapter 6)
3. Technology demonstration (see Chapter 7)
   (a) Sensors
   (b) AI package
   (c) Agent core
   (d) Toolkits
   (e) Supplementary infrastructure
4. Literature search (existing systems, theory of operation, AI techniques, mathematical background, requisite graph theory) (see Part 2)
5. Run logs, experimental quality of service (QoS) data, and analysis (see Chapter 8)

Part II

Literature Review

Chapter 2

Artificial Intelligence (AI)

Significant research is required to design and implement the Net.Sense vision. Such a system encompasses many areas, ranging from the fundamental ("What, exactly, is an agent?") to advanced ("How can many simple agents cooperate cheaply?" or "How should distributed data structures address write-locking?") This chapter provides background material relevant to the design, plans, and goals for Net.Sense systems. As we shall see in the last chapter, there are several areas where future research is needed before future development can continue.

2.1 The (Multi)Agent Paradigm

Definitions of agents abound. Maes defines an agent as "a process that lives in the world of computers and networks and that can operate *autonomously* to fulfill a set of tasks" [73]. Some say they are autonomous, intelligent, social, diverse, honest, and collaborative [101, p4-5]. Yet others feel that agents must be autonomous, social, reactive, and proactive. From time to time, other attributes (mobility, veracity, benevolence and rationality are attributed to agents as well [109, p2].

Perhaps because of their generality, we will satisfy ourselves with the first two definitions. For our purposes, agents are pieces of software that are autonomous, social, and reactive. (Proactivity will come later.)

2.1.1 Agent-ness

Autonomy

One of the most significant difference between agents and everyday programs is agents' ability (exercised or not) to consider and evaluate commands sent to them [60]. This consideration and evaluation effectively changes commands to requests or propositions. Adding such a 'consideration' layer does not eliminate the possibility of treating requests as commands. (A very simple layer taking a "Do whatever is asked of you" approach is proof enough of that.) However,

treating commands as requests creates a more straightforward design process for some types of problem domains. In particular, modelling distinct, self-interested entities (especially sub-roles often filled by humans) appears to be simplified by the agent paradigm. This is because the designer can put action decisions in the agent, preserving and strengthening the analogy between the modelled real-world and the model agent-world.

An interesting follow-on to the autonomy criterion is that of identity. While agents may interface with their environs as closely as we program them to, we believe that true autonomy requires self-containment and distinction from its environment; in short, a distinct identity. By self-containment, we mean that the agent is sufficiently distinct from its environment and that, if desired, the agent *could* be separated and moved in its entirety. Note that this is *not* a mobility requirement. Agents may remain immobile for the duration of their existence, but for our purposes, the identity criterion requires that the agent know of all its parts and extents. Further, should separation be necessary (i.e. for movement), the entirety of the agent can be distinguished and separated from its environment.

Sociability

Sociability requires that agents have the capacity to transmit and receive messages, whether or not they exercise it. These messages may come from any source, and may be bound for any destination, but must be in a format understood by all acents involved. Languages such as the Knowledge Query Markep Language (KQML) and Agent Communication Language (ACL) [42, 26]. As a general rule, agents interact with each other over the course of their lives. Agents are not mandatorially social. However, much of the power arising from the agent paradigm is grounded in rich inter-agent interactions [100, 110].

It is worth noting that sociability does not imply benevolence (non-conflicting goal sets, implying that agents try to do what is asked of them) or veracity (truthfulness) [109, p2]. For us, sociability only requires that agents be *able* to exchange messages regardless of whether they do so or what those messages contain.

Reactive

Simply put, reactivity means that the agent can sense and react to its environment. Such sensation and action may be extremely primitive, (for example, witness paramecium's dart-and-tumble behavior [62, p96-97]) but without it, the agent has no circumstances in which to be autonomous. Without external stimuli of some kind (be they messages from other agents, sensory data, etc) the agent has no data that could challenge its preprogrammed course of action. It is reduced to a mere program.

2.1.2 Applications

Multiagent systems (MAS) have been applied to countless problems. In the academic realm, they have been used extensively on the Travelling Salesman Problem (TSP) [31, 104, 65] and general pathfinding problems [99]. In the everyday realm, MAS techniques are seen in email clients [17], web browser assistants [29], automated cleaning [102, 103], as well as more specialized areas including humanitarian demining [35], undersea search [13], and medical decision support [59].

2.1.3 Competition and Collaboration

Background & History

Historically, agents 'consciously' operating in groups (that is, conscious of their group membership, not of themselves per se) have tended to interoperate in either a collaborative or a competitive manner. Collaborative behaviors range from the unknowing [87] to the deliberate [63], from the simple and unplanned [99] to the complex.

Application

Given the wide applicability of the two interaction styles, it seems pointless to cite specific examples of one style in application over another. Rather, it is believed that the 'right' interaction model is the model that best reflects the system under consideration. In general, competitive behaviors are viewed as more suitable for optimization problems, and collaborative behaviors as more suitable for potentially or probably insoluble problems.

Competitive behaviors seem ideal where optimization, rather than solution, is the goal. Competing for scarce resources, bidding on contracts, and inter-agent bargaining direct computational resources towards performance of some emininently completable task with the minimum expenditure of resources.

Collaborative behaviors, on the other hand, seem more suitable in situations where emphasis is on finding *any* solution to the problem at hand, or where a 'contract provider' would be a needless contrivance. Collaborative behaviors place emphasis on contributing services or distributing and performing subtasks to achieve a common, difficult, if not insoluble, goal.

Because one can rephrase essentially any question for either style, this discussion is somewhat academic. Optimization problems can be rephrased as "Can this problem be solved at x cost?" where x begins small and is increased until an affirmative answer is given. Near-insoluble problems can be rephrased for competitive systems by issuing contracts for subtasks that agents believe are within their power to solve.

Net.Sense Connection

Net.Sense must address problems at many scopes to meet its functional goals. Some goals (for example, "maintain system security") are extremely complex, requiring nontrivial reasoning effort on the part of multiple expert agents to detect and assess threats, and propose solutions. Collaborative behaviors are in order for such problems. On the other hand, problems such as collecting data from machines are clearly tractable, and hence our interest is in minimizing the resources required to provide a given quality of service. In such a situation, the ideal solution would seem to be setting agents to forage the network for data.

We envision domain-expert agents holding virtual 'debates' over proposed courses of action, with the goal of the debate being to select the minimum cost solution that meets the given operational requirement. For example, a security agent may have indicated a need to screen out traffic from a given host. An agent responsible for managing Network Interface Cards (NICs) may propose turning down the interface. This proposal would be unacceptable to a web server agent, which would be unable to perform its task were the proposal implemented. In the end, a firewall agent's competing proposal to add a 'drop' rule may be selected as the best compromise between operational necessities and mission requirements.

2.2 Agent/Community Control

2.2.1 Hardcoded/Scripted

Background & History

Perhaps the conceptually simplest agent control model is the one most closely related to a familiar activity: procedural programming. Some task is to be performed, a task can be accomplished by performing some sequence of predefined, coarse-grained steps. In the literature, [33, 77] these sequences are called scripts. For our purposes, the predefined steps available for scripting do not afford the power of a rich and expressive language like C++ or Java; rather, primitives deal with domain-specific concepts [97, p2]. For example, a bank's lending agent might have a domain-specific language (DSL) with primitives for cash flows, balances, and so forth [8]. Outside their DSL, the scripted agent has a severely attenuated (if at all existent) view of the world.

Application

Beyond the financial field, DSLs have application in software engineering [94, 85], telecommunications [68, 32], data-structure programming [90], and even board games [86].

Despite the comparatively crippled nature of scripted agents, they have a variety of uses where the benefits of simplicity outweigh needs for expressive power and fine-grained control. By capturing a domain's primitives and actions in its own terms, the simplicity of DSLs can place significant power in the hands

of domain experts even if these experts have little or no programming experience. Agents controlled by DSLs place most of the power of the agent paradigm close to where it's needed: at the user's fingertips. In more advanced user environments, DSLs can help reduce programmer faults and decrease development time [85].

Net.Sense Connection

While Net.Sense is intended for a technically adept user base, it is felt that DSLs and scripted agents would provide significant benefit to Net.Sense users. Specifically, it is believed that use of scripting and/or DSLs in Net.Sense should reduce the time to develop and deploy agents while reducing the frequency and severity of faults discovered within user-developed agent behaviors.

2.2.2 Belief-Desire-Intentionality

Background & History

The Belief/Desire/Intention (BDI) architecture is a tool for designing 'rational' agents. 'Rational' means that agents' actions follow some organized series of intentions toward acheiving one or more goals. Intentions are courses of action. They are determined by the desired world-state, beliefs about the (present) world-state, and (semantic) beliefs relating actions and their effect on the world-state [84].

For example, an agent might have a desire to eat donuts, and a 'present' belief that the agent has no donuts. The agent now has a problem, namely that it has no donuts but wants some. Giving the agent a 'semantic' belief that donuts can be purchased from a vendor leads to an intention (and thence, a plan) to go to the vendor and obtain donuts.

Intentions may be determined via a variety of methods. Agents may be utilitarian[1], greedy[2], or may use other criteria, such as the "maximin" heuristic[3].

Beliefs can be represented in both discrete and continuous fashions. Boella *et. al.* evaluate goals and plans on the basis of a real-valued utility function [15], but it is equally feasible to operate on the basis of discrete utility values.

BDI architectures can easily control local behaviors, provided that the agent's desires are consistent (non-conflicting). Agent motivations can vary in complexity from a desire not to revisit spaces to intricate evaluations of future, strategic positions. Implementation can be as simple as an inference engine loaded with rules describing the agent's beliefs, desires, as well as rules for generating intentions.

BDI systems suffer from mobility issues arising from the discontinuity in agents' environment, and the resulting conflicts in their belief and intention bases [24]. Systems with real-time requirements (air-traffic control, power-plant

[1] "Do what results in the maximum value."
[2] "Do what maximises the value I get."
[3] "Choose the intention with the best 'worst case' scenario."

control, etc) or extremely frequent belief updates are also unlikely to operate successfully, crippled by an inability to reevaluate available intentions within hard time constraints.

Application

BDI/rational agents have been used in military applications including fighter control [57], anti-air defense [78], telecommunications management [56], office assistance [11], museum tours [23] and even spacecraft control [80]. The generality of the BDI architecture provides applicability across a variety of problem domains.

Net.Sense Connection

As stated above, the BDI agent architecture describes agents as rational entities, each with their own goals, knowledge, and ability to examine and change the world in which they operate. This architecture is relevant to Net.Sense in two ways. First, actor agents that Net.Sense dispatches to resolve problem conditions may well be designed and programmed from a BDI viewpoint. Far more relevance is revealed when looking at a Net.Sense system with a wide-angle lens: Net.Sense turns the managed network into something resembling a large, overarching BDI agent. Like the people in Hobbes' *Leviathan*[44], agents in Net.Sense communities together form something larger, together achieving greater purposes than any member could alone.

It is believed that BDI agent architectures are particularly applicable where agent complexity varies significantly. BDI designs support "heavy" agents with extensive knowledge and advanced self-tasking mechanisms. However, the BDI concept remains useful when designing "light", reactive agents. Such agents can be implemented frugally by eliminating the overhead associated with desire and intention modification. These simple-minded agents lack the flexibility and complex interactve character of heavier agents, but their small size makes up for this by enabling them to be deployed in greater numbers.

2.2.3 Biologically-Inspired

Background & History

MAS developers can learn from biology in at least two areas: population control and coordination. To maximize scalability, direct communication and interconnection between agents must be minimized. Nature provides many examples of indirect communication through chemical markers. While no purely software MAS will be able to produce chemical markers, simulating these chemicals in the virtual space may prove rewarding.

Population control

Biology is rife with examples of independent 'conscious' agents operating independently or in groups. From amoebae at bottom of the animal kingdom

[62, 98] to mosquito swarms and wolf packs, nature presents us with heuristics and methods for agent control.

Perhaps the conceptually simplest of these heuristics is the 'energy' heuristic that regulates agent population. Agents move about their environment, collecting and consuming resources. These resources (for example, plant life) are generated by the environment at a given rate α, and are transformed into energy agents at some other rate β. When an agent has acquired a certain amount of energy, it may reproduce, cloning itself and dividing its energy between it and its offspring. Conversely, when an agent runs sufficiently low on energy, it starves to death.

In traditional population modelling [38] we see that the α and β above dictate the environment's 'carrying capacity', how many agents can survive in the long term in a given environment.

Coordination

The simple tropic principles of attraction and repulsion have been used successfully by nature for millennia. Even the simplest life forms display effective behavior. For example, *E. coli* and paramecia detect toxins in their environment and respond by randomly changing direction and moving, thus evading the danger [62, p96–98]. Still other single-celled organisms (*Dictyostelium discoideum* and slime molds) emit chemical 'distress signals' when food is scarce [62, p96–98]. These amoeba-like creatures detect these distress signals, reemit them in an autocatalytic process, and move towards their source. On meeting, they are subsumed into a multicellular slug-like organism that moves as a unit to a more favorable location. The 'slug' then releases spores, that become new amoebæ[62, p98].

More advanced forms of life also display tropic behavior. Termites follow pheromone trails [66], and flies are attracted to the smell of rot. Bees 'remember' what the areas around landmarks (favorable feeding sites, the hive, etc) look like [111].

Application

Pseudorandom movement and tropisms can be exploited in a variety of scenarios. Positive tropisms lead to efficient results for path-finding [31] and trail-building [99]. Negative tropisms have produced results useful in autonomous cleaning [102] and graph covering [103].

Net.Sense Connection

Population control

The goal of resource parsimony requires that a Net.Sense system contain the minimal number of agents. In a sufficiently large system, it is felt that the sensor agent population will be far larger than their thinker and actor counterparts. As such, a flexible, automatic method of population control is needed. Where nature's method leads to a larger population of creatures who live at or about the subsistence-level, our goal is to minimize resources consumed. As a result,

a Net.Sense MAS agent population should stabilize at a minimal number of 'maximally fed' agents.

Tropisms

Net.Sense addresses the predictability problem by adding randomness to agents' movement. While solving the problem, this creates issues of efficiency and responsiveness. If agents move on a purely pseudorandom basis, they will spend time in areas of the network where they are not needed. Further, purely pseudorandom movement does not allow Net.Sense to direct 'attention' when such control may be desired, for example, towards the site of unusual network traffic.

It is proposed that these issues be addressed by the introduction and manipulation of agent tropisms. By permitting thinker agents to create and control the strength of sensor agent tropisms, the system should be able to direct sensory attention around the system as needed.

In addition to providing control over the sensor community, tropic techniques could be used for other, more administrative purposes. For example, by marking nodes visited when migrating across network segments, agents could dynamically update and follow optimized routes through extremely large or dispersed networks for minimal cost.

2.2.4 Subsumption

Background & History

The subsumption architecture was developed for use in real time, autonomous robot control systems [20]. It addresses primary issues in nontrivial robot control (timeliness and complexity) by turning the default control paradigm on its ear.

Traditionally, the basic control loop for a robot ran sequentially along the lines of "Perceive, model, plan, execute, actuate" [20, p1]. These steps and the interfaces between them became more and more complex, and ultimately posed scalability (in terms of computational time) and implementation (in terms of developer time) issues. The internal models required to support traditional reasoning mechanisms were unwieldy, constraining, and demanded impractical detail and accuracy. Brooks introduced an architecture based on parallel independent behaviors that can override each other in a process called 'subsumption' [20, p7]. Arguing that "the world is it's own best model" [22], Brooks' argues that agent systems should maintain no internal state, but instead react to the world directly as it is presented.

The subsumption architecture differs from traditional control design in three ways. First, (robot) agent functionality is decomposed into independent behaviors rather than sequential tasks. As such, sensor data are simultaneously fed to all such behaviors. Second, inter-behavior connections are minimized. Individual behaviors outputs are connected to the (robot) agent's actuators in such a way that some behaviors can override others, thus creating a 'layered' architecture. Thirdly, complex agent-wide performance is improved by incrementally adding new behaviors [25].

These three differences can provide significant and powerful advantages in (robot) agent control. The timing issue is handled by the simplicity and independence of the behaviors, and by provision of simple, cheap, 'safe' behaviors at the lowest levels of the hierarchy. If higher-level (more sophisticated) behaviors produce timely results, they can override the less-sophisticated behaviors. If higher layers fail to respond in time, the (acceptable if sub-optimal) lower-layer behaviors will control the agent [20, 21]. Subsumptive architectures can easily be made more robust by adding redundant layers, and their reactive nature and low reponse latency makes them resilient to highly dynamic environments [25].

Like all things, subsumption designs have their limitations. Subsumption shines in the area of scalability, but this scalability comes largely in parallel hardware module implementations [20, p3–4]. Implementing complex system-wide behaviors can require a set of many subbehaviors that can be difficult to organize for subsumption without introducing unwanted side effects. (It appears [13] that adding a metalayer that selectively activates sets of behaviors (like roles) can usefully reduce these interactions.) Brooks' robots, however, are 'greedy' in the sense that they base their actions off of immediate, 'local' data only.

Application

Brooks constructed a robot (Herbert) that finds and discards soda cans [22, p5]. Instead of a complicated set of inference rules or a trained neural net, Brooks programmed a series of independent, event-driven modules that certain events trigger. One module avoids collisions, while another attempts to get in front of soda cans. A third is responsible for picking up things that are directly in front of the robot. Together, these modules provide remarkably plausible and resilient behavior [62, p116]. Robots such as Toto [76], the Sea Squirt AUV [13], and of course Brooks' robots at the MIT AI lab [20, 21] demonstrate the viability of the subsumption architecture.

Net.Sense Connection

Net.Sense effector agents must often balance orthogonal or conflicting goals. For example, an agent might (simultaneously) have the following goals:

1. Verify packet filter firewall (PFFW) rules on machines W, X, Y, and Z.
2. If necessary, modify PFFW rules to meet given requirements.
3. Avoid visiting compromised hosts.
4. Avoid visiting hosts with load > 4.
5. Minimize user risk associated with modifying firewall rules.

This goal set contains several potential conflicts. For example, suppose the agent is attempting to modify host X's firewall rules, but on arrival finds that

root is logged in via ssh. The agent may evaluate the risk associated with disconnecting root's session and determine to return at a later time, send a message requesting root to log off temporarily, or announce failure to meet the current goal instance.

The subsumption architecture has value for 'soft' conflicts as well. 'Soft' conflicts involve balancing competing interests or goals. For example, one might wish to minimize sensory agent count, agent footprint, and bandwidth consumed in the collection process. Augmenting individual agents' sensor loadout can reduce the number of agents, but bandwidth requirements go up as agents now carry along infrequently-used sensors at every move.

The proposal is to use subsumption techniques to help optimize Net.Sense's behavior in situations with interconnected or conflicting controls. It is also felt that such techniques would increase Net.Sense systems' robustness in the face of highly dynamic operating conditions.

2.2.5 Swarm Intelligence

Background & History

Swarm intelligence is a relatively recent development in the Artificial Intelligence family tree, but it has spawned a variety of techniques, algorithms, methods, and potential applications. Whether one's method of choice is based on ants [104], termites [66], or 'particles' [62, p287], the core principle is largely the same. In the plainest of terms: " 'Small, dumb, and plentiful' can succeed where 'large, smart, and few' cannot." For the most part, all such methods together are lumped together as we are not as much concerned with their details as their principle and the properties that result from implementing it.

Relevant Qualities & Applications

Survivable Swarm systems are, almost by definition, highly distributed systems. From an operational standpoint, this means that they are highly survivable in the face of localized disruptions. Should agents in one area be damaged or destroyed, their identical brethren (located elsewhere) will continue operation. This property is highly desirable for systems intended to be deployed in a hazardous environment, be that a minefield [35] or (as intended in the thesis) a potentially malfunctioning or compromised computer network.

Degrades Gracefully In addition to surviving, swarm systems degrade gracefully. That is, system performance is smoothly correlated with the degree of damage the swarm has taken. The highly distributed, parallelized nature of swarm systems provides a high degree of redundancy and uniformity in the swarm's operation. Should a member of the swarm be taken out of operation, the work that agent previously performed will simply remain in the global 'task pool', where it will automatically be performed by other agents in the swarm.

Individually Chaotic Individual swarm members are not possessed of a great and unifying plan that tells them where to go and what to do. Rather, members of a swarm move semi-randomly until external stimuli force the agent into a particular task, path, or action. This lack of organization provides significant benefits for systems like Net.Sense, which wish to balance load across hosts as well as time. As an added bonus, the variability in agent paths can confound potential attackers' attempts to evade detection by Net.Sense monitors.

Net.Sense Connection

Swarm systems seem to be everything one could want in a defensive data-collection system. Swarm agents require little to no operational supervision, require minimal control code and memory, and provide behavior 'patterns' that deny attackers known safe windows in which to operate. At the same time, this randomized behavior can provide statistical metrics about quality of service experienced by the managed network. They are relatively lightweight (having no complex AI), are highly survivable, degrade gracefully, and are easily replenishable owing to their interchangeability.

We plan to exploit these characteristics in the sensory arm of the Net.Sense system. Light agents should swarm randomly over the managed network, collecting and reporting data. Should agents be lost or destroyed, service quality should decrease, and new agents can be automatically spawned in their place. The random nature of their movement should increase attackers' difficulty in evading detection, while system operators retain the ability to make useful, verifiable statements about system service levels.

The astute reader may wonder why we have chosen to take the swarm movement approach instead of a randomly chosen pattern. In the search for predictable, repeatable service, the first step might well be to randomly program or evolve a fixed route that satisfies the user's requirements. Indeed, such solutions perform well for cleaning applications [102, 103]. However, it is felt that these solutions are inadequate for a security tool because their predictable nature could permit an attacker to organize and plan an offense. Net.Sense attempts to solve the predictability problem by continually randomizing the movement of swarm members. Doing so destroys the ironclad predictability that system administrators currently enjoy with periodic schedulers such as cron. Net.Sense fills the gap with statistical assurances about the quality of service (QOS) experienced by the managed network.

2.3 AI Techniques

2.3.1 Traditional/Reasoning AI

Background & History

Inference engines (also called production systems) consist of a set of production rules, a fact base, and controls that govern the application of those rules. These systems operate by creating 'productions'; statements of the predicate calculus that describe the system they model. These productions are created by applying predicate calculus rules called 'production rules' to their fact base as directed by their control code [72, p171].

Inference engines come in two flavors, according to the way in which they navigate the state space of potential predicates. When a new fact is added to the fact base of a reactive (data-driven) system, the control system recognises it and fires the inference rules that involve that fact. The resultant productions are added to the fact base, causing a cascade of updates that adjust the system's model to reflect the effect of the original new fact. Compare this with a proactive (goal-driven) system, where the system starts with a desired goal (a theorem, or other desired production) and proceeds to work 'backwards' (towards elements in the fact base) until actual facts are reached [72, p182].

Inference engines play a key role in the arena of expert systems [72, p210]. Expert systems consist of a knowledge base, a suitable inference engine, and a user interface. These systems are equipped with knowledge bases that enable them to reason (in limited fashion) over problems with the knowledge of an expert in their domain. While these systems lack creativity and are often limited to narrow domains, they have shown to be of significant value and are thus widespread in their application.

Application

Expert systems lend themselves well to problem diagnosis, and as a result much work has occurred in the medical field. Systems include MYCIN [88, 2] (which diagnosed spinal meningitis), PUFF [4], (pulmonary problems), TERAP-IA (adult pneumonia treatment) [10, 36], and Dr. Gait (gait issues for cerebral palsy patients) [58], to name a few.

More relevant to Net.Sense is work with intrusion detection using expert systems. Most systems in this area (such as EMERALD [82] and (N)IDES [7]) integrate statistical anomaly detection with existing expert system technology to attempt to detect intrusions.

Net.Sense Connection

It is believed that a community of domain-expert agents is an ideal design metaphor for both detecting aberrant operating conditions and managing and balancing the actions of the Net.Sense system. Consider an expert agent such as a traffic manager detecting lost packets on a critical network link. The

expert system would dispatch agents to collect additional data, which perhaps indicates a distributed denial of service (DDoS) attack. The 'quick answer' may be to simply drop the inbound interface, but doing so may cause unacceptable conflicts with agents who provide other services, such as external web presence. Dialog between agents could result in acceptable compromises, for example, the two experts may agree to notify the upstream provider to drop inbound traffic except from certain customers and privileged database clients. While not eliminating the DDos problem entirely, the compromise preserves core services while restoring some degree of network performance.

2.3.2 Ontologies & Knowledge Representation

Background & History

Ontologies, like agents, are different things to different people. Some view an ontology as a "catalog of types"[92], others as an "explicit specification of a conceptualization" [54, p2], or "a particular conceptualization of a set of objects, concepts and other entities about which knowledge is expressed and of the relationships that hold among them" [67, 2]. They "are structured vocabularies representing the schematic metadata of a particular application domain"[46, p2]. While the definitions differ in the details, at the core of each is the idea that an ontology is a description (of whatever sort) of a knowledge domain.

Formalizing knowledge domains in this manner enables programs (in our case, agents) to communicate meaningfully. Consider the problem of web-searching. Ignoring Google and other traffic-tracking engines for the moment, there are two prevailing methods for indexing the World Wide Web (WWW). The first method is manual page examination and classification, resulting in the creation of a directory that users may browse. This greatly reduces the number of irrelevant results returned, at enormous financial expense and time as humans must review (and keep reviewing!) each of the billions of pages on the web. The second method amounts to rapid string matching. This has the advantages of being highly automatic, but search quality becomes highly dependent on query term choice and spelling. Searching for a term in one language may not return useful results written in another language, because the nomenclature for the search target differs across languages.

With that in mind, consider the problem of communication among software agents. Like human agents, software agents must communicate in common (or at least, translatable) languages. Two such agent communication languages (ACLs) are KQML, the Knowledge Query Markup Language [43] and the FIPA (Foundation for Intelligent Physical Agents) ACL [26]. These languages describe the syntax and semantics of languages that enable agents to assert facts, query other agents, request actions, and generally communicate.

Many, if not most ontologies currently in existence were developed for the purpose of facilitating MAS construction. Most MAS require the ability to either communicate among themselves or reason over information they acquire. Some ontologies are implicit, meaning they are coded directly into the programs

that use them. This can be simpler than specifying the ontology explicitly, as the overhead of creating, managing, selecting, and using the various ontologies is eliminated or vastly reduced. This simplicity comes at the cost of flexibility, expandability, and maintainability. Explicit ontologies, on the other hand, require significantly more effort but are flexible, reusable, and maintainable. Explicit ontologies are practically a requirement when agents from multiple sources interact, as the ontology serves to define the language the agents will use to do so.

Ontologies describe the context within which agents should interpret messages. Without an ontology, the messages "attack at dawn" and "host X's load is 5.31" have the same meaning: no meaning at all. An agent with a military ontology (perhaps containing temporal concepts as well as military maneuvers) could interpret the former message as it was presumably intended. If a management agent had both military and system management ontologies, it could be programmed to interact with the military agent. For example, the management agent could send a 'retreat' message should system load exceed 12.

If the KQML and (FIPA) ACL specifications are similar to grammar textbooks, then ontologies are similar to miniature dictionaries. Where KQML and ACL describe how to say, ask, etc., ontologies provide a domain's concepts, terms, and the relations between them. A sandwich ontology, for example, might contain noun concepts ('roll', 'knife', 'butter',), verb concepts ('stack', 'spread', 'cut'), type information ('rolls' are a type of bread), and relations between these concepts('bread supports vegetables', 'knife cuts bread').

Even now, our example cannot describe a simple sandwich! This is because ontologies contain classes, not class instances. An agent with the sandwich ontology 'knows' *of* breads, but has no bread with which to compose a sandwich. Instances or realizations of the ontology concepts are part of the agent's fact (or knowledge) base, and might, for a given agent, consist of a number of slices of ham, wheat bread, and mayonnaise.

'Noun' classes' semantics are their attributes (also elements of the ontology) and their associated (inter)actions. In short, a (type of) thing is defined by its interactions with other (types of) things. 'Verb' classes, then, are defined by the changes they bring about, which we can represent as pre- and post-conditions over other classes in the ontology. That is, the action's *change* is its meaning.

Application

Many MAS use ACLs to communicate. Theoretically, agents speaking the same language shoud be able to hold minimal communications. Thus far, however, even ambitious projects such as AgentCities [45] have realized little inter-domain interaction. This is understandable in light of Agentcities' young age (the project began in October of 2001) and the significant challenges involved in interfacing ontologies. As these ontologies are primarily manually created, system specific endeavors, they are time-consuming and costly to develop and employ.

This cost, coupled with the vastness and popularity of the WWW, has added impetus to a movement towards a 'semantic web'. The semantic web is an "ex-

tension of the current Web in which information is given well-defined meaning, better enabling computers and people to work in cooperation" [70]. The over-simplified project goal is to develop and promote a common way for content authors to describe their content and its relations in order to facilitate (automated) manipulation of / interaction with their content. In order to work, a great deal of domain knowledge needs to be made available and standardizable. This is the task of the W3C Ontology Working Group [53]. With a semantic web (and supporting ontologies) in place, agents-in-the-web would have a fantastic library of information to draw on and interact with on the users' behalf.

Of course, fields other than MAS have seen significant ontology research. Owing to the massive amount of medical literature, and the importance of timely access thereto, efforts to develop and link ontologies for medicine [81, 50, 41, 98] are abundant. Other research areas include translation [74, 75], business [37, 95, 49], and natural language processing (NLP) [16].

Net.Sense Connection

As a network management tool, Net.Sense exists largely to collect and analyze data. Net.Sense' distributed organization requires agents to communicate and evaluate many things, including sensor readings, requests, commands, and the managed network's state. To enable Net.Sense' automatic response capability, the AI that drives Net.Sense must be able to 'understand' the effects that toolkits will have on the network. These capabilities require a common language (both 'grammar' and 'terms') to facilitate inter-agent communication. Ontologies are one of the knowledge-representation tools that make this possible.

Once suitable ontologies are in place within Net.Sense, agents should communicate via the FIPA ACL to share and discuss sensory data, evaluations of the managed network state, and courses of action which will become command sets issues to actor agents.

2.3.3 Blackboarding

Background & History

Blackboarding [72, p196–198] (sometimes called whiteboarding) is a collaborative methodology that calls for a variety of specialized agents to simultaneously examine data and process it to the gain of other collaborators. Agents may revise their own results in light of new results from other agents, and as time passes the group will (hopefully) come to consensus on a correct answer to a particular problem. Multiple lines of reasoning and search techniques may be independently pursued and opportunistically utilized by other agents. The qualities have made blackboard techniques popular for hard or potentially insoluble problems [28].

Traditional blackboard systems operate by posing a problem's specifics onto a shared space, called the blackboard [72, p196]. Independent knowledge sources (KSs) detect modifications to the blackboard, evaluate the changes, and update

their published results accordingly. In this way, updates cascade until either a solution is found or new data is required. By exchanging partial or tentative solutions, KSs for each problem subdomain may assist each other in selecting portions of their search spaces for pursuit.

For example, consider a blackboard system handling logistics for a combination factory and warehouse. Suppose a KS interfaced with a shipping contractor indicates that the refrigerated trucks scheduled to collect finished material will not arrive for an additional week. A KS responsible for managing cold storage space would, based on the new data, revise its estimates about the free floor space available. The KSs responsible for production may in turn need to rework the current production schedule to accomodate the delay, perhaps by switching to a product that does not require refrigeration.

Application

One of the first blackboard systems was HEARSAY-II [40], a speech understanding program. It and its successor (HEARSAY-III [39]) were enhanced by the ability to request additional processing on unexpected results [72, p196-198]. Since then, blackboarding (in various forms) has been applied to problems such as multiple-goal management [71], agent control [28, 34], knowledge reuse [83], music transcription [61], and data interpretation [48].

Blackboard technology has spawned two developments we find particularly noteworthy: Linda and JavaSpaces. Linda is a programming language [93], developed for parallel processing by an eponymous group at Yale. The language is based on the concept of a 'tuple space'. A tuple space is similar to a corkboard in that users may add and remove data items posted to the board. When data is removed, it can be manipulated and reposted in a manner resembling that of a database transaction.

Programming with Linda's tuple space concept can be similar to performing blackboard-style interactions. Many of its underlying concepts can be found in JavaSpaces [47], a recent product of Sun Microsystems that provides a framework for constructing tuple space-like entities called JavaSpaces.

Net.Sense Connection

We are interested in blackboarding as a tool for data management in addition to domain-expert collaboration. Net.Sense's InfoSpace can be seen as a shared dataspace [79, p6]. However, where the traditional view sees the dataspace as the sole means of communications between agents, Net.Sense views the InfoSpace more as an active data repository. The InfoSpace provides a robust information storage and manipulation space for Net.Sense reasoning and command agents.

It is believed that the InfoSpace should enable Net.Sense to more effectively manage information and support decision-making at the highest levels. It is felt that the flexibility afforded by a blackboard-inspired architecture should enable more synergistic reasoning and workflows between the sensor, domain expert, and interface agents that comprise a Net.Sense system.

2.3.4 Genetic Algorithms

Background & History

Genetic algorithms (GAs) are search strategies adapted from concepts [27, p476-481] in biology and evolutionary sciences [52]. Elements in the solution space to be searched are represented as arrays of individually-manipulatable 'genes', such as integers or bits. The search is initialized by randomly generating a number of elements from the search space. The 'goodness' or 'fitness' of a particular element is determined by a human generated fitness function. A percentage of the fittest elements are retained for 'breeding', while the remainder are discarded. The retained elements are paired up and 'mated', exchanging and/or mutating genes. The resultant (offspring) elements are then evaluated for fitness, and the cycle repeats [51, p7-11]. The details of the mating process vary from researcher to researcher, and from paper to paper, but a good introduction can be found in Chapters 3 and 5 of [51].

GAs have two qualities worth noting: operator independence and satisficity.

Operator Independence Once suitable fitness functions have been defined, GAs essentially execute without supervision. This independence, however, requires careful and robust selection of termination criteria. Because GAs essentially search for optimal problem solutions [51, p6-7], an overly demanding termination condition will needlessly consume cycles trying to reach an unattainable goal. If, on the other hand, we set our sights too low, the search may terminate while progress can still be chearply obtained.

Satisficity GAs are good at cheaply determining solutions that are close to (but not always) optimal [51, p7]. Unlike pure theoretical problems, the practical problems Net.Sense is concerned with are much more amenable to approximate solutions. Because "good enough for the operator" is good enough, GAs' approximate solutions pose no obstacles for their use within Net.Sense.

Application

Genetic algorithms have seen use in a variety of application areas. These areas include physical database design [96], manufacturing composite laminates [91] and flywheels [9], op-amps [14], circuit design [64], the travelling salesman problem [6], Bayesian networks [69], and neural networks [5, 30].

Net.Sense Connection

Net.Sense agents may be equipped with a variety of sense/effect toolkits. The problem of choosing the best toolkit combinations is, at worst, exponential in the number of available toolkits. At this time, the improvement over the average case is unclear. While it is clearly feasible to manually specify customized loadouts for a small agent population, it is clearly infeasible for large populations.

These loadouts need not be perfectly optimized as they can be modified at runtime. However, by approximating the correct loadout, the time and bandwidth costs of equipping an agent (community) can be amortized over time, smoothing out the footprint that Net.Sense incurs.

2.4 Review

In this chapter, relevant AI techniques suitable for use in a Net.Sense system have been discussed. The discussion covered their nature, history, applications, and potential role in a Net.Sense system.

Chapter 3

Mathematics

This chapter reviews some of the mathematical background that will be required for either conceptual understanding or concrete implementation of a Net.Sense system. These concepts include basic graph theory, randomized graph covers, and distribution-free statistics.

3.1 Graph Theory

While Net.Sense is not primarily a graph-theoretic tool, some basic theory helps to establish concepts required for later discussion.

3.1.1 Background & Theory

Recall that a graph G consists of two sets, V and E. V is the (non-empty) set of vertices or nodes in G, while $E = \{(v_i, v_j)\}$ is the set of all undirected edges connecting nodes in V [107, p10].

Graphs, as sets, have no formal 'shape'. However, shape and well-defined space is quite important in our daily lives, and thus it is reasonable to wonder how something analogous would affect Net.Sense's operation. The analogue is based on adjacency. In addition to fleshing out this concept, notation to describe graphs with particular shapes is introduced.

As previously mentioned, the set-based nature of graphs implies that they are shapeless. There are therefore an uncountably infinite number of ways to draw a graph given only the sets V and E. However, consider the adjacencies and distances implied by edges in E and use them to describe 'neighborhoods' of vertices.

Let us begin with simple graphs. First consider a graph of n vertices $V = v_1, \ldots, v_n$ and $n-1$ edges $E = (v_1, v_2), (v_2, v_3), \ldots, (v_n, v1)$. This graph is simply a 'ring' or 'cycle' graph in which each node in it connects to exactly two neighbors (it is 2-regular), therefore it is called C_n, the Cycle graph on n nodes

[107, p36]. Though C_n can be drawn many, many ways (some very un-ring-like), it is said that its 'shape' is circular, ring-like.

The next graph is called K_n, the 'fully connected simple undirected graph' on n vertices [107, p36]. We again have $V = v_1, \ldots, v_n$, but this time E contains all possible edges. So, $E = (v_1, v_2), (v_1, v_3), \ldots, (v_1, v_n), (v_2, v_3), (v_2, v_4), \ldots$. Note that because edges are undirected (meaning that $e = (v_a, v_b)$ can be traversed both from v_a to v_b and vice versa), the cardinality $e = |E| = n!$. Subgraphs of the K_n form are commonly called 'cliques' [89] because all vertices in the subgraph 'know' each other. Since all nodes in K_n are connected to all other nodes in K_n, they are all adjacent to each other, and therefore comprise one neighborhood.

At this point some informal terms specific to this thesis are introduced. Given a vertex v in G, the 'neighborhood' of v, denoted $N(v)$, is the set of all vertices that are adjacent to v. More formally, $N(v)_{x,y}$ because any node on the grid can be identified with two integers, like Cartesian coordinates. Connecting the top and bottom edges of $Z_{x,y}$ (connecting the top and bottom edges), gives a 'cylinder' $Y_{x,y}$ where $v_{i,0}$ is adjacent to $v_{i,y}$ for all i, and all nodes are now 4-regular except for the two cycles at the ends of the cylinder.

Adjoining the cylinder ends so that $v_{0,i}$ is adjacent to $v_{x,i}$ for all i produces a torus or doughnut shape $T_{x,y}$, where all nodes are 4-regular.

Now we can begin to use these basic graphs. The most interesting thing that is done with these graphs is to 'walk'[107, p34p] on them. A 'walk' W on a graph G is defined to be a succession of edges e_i linking adjecent vertices:

$$W = ((v_a, v_b), (v_b, v_c), \ldots, (v_w, v_x))$$

This succession is shorthanded as "abc ...wx". If no vertex v_i is ever visited twice (that is, there exist no two pairs in the succession that begin at the same vertex) the succession a 'path' [107, p35]. If a walk ends on the same vertex it started on, it is called a *closed* walk [107, p35]. A cycle is a 'closed' path – that is, a path whose last edge connects back to the starting vertex. The length of a path or cycle is the number of edges it contains [107, p35].

3.1.2 Application

Graph theory has applications from circuit design to cryptography to cartography. As a branch of mathematics, a taxonomy of its uses and applications are far beyond the scope of this paper.

3.1.3 Net.Sense Connection

Graph theory is relevant to Net.Sense because it enables meaningful discussions regarding the motions of agents on the managed network, as well as the organization and layout of managed networks. To lend rigor to statements regarding the quality of service Net.Sense can provide, we must be able to speak precisely about the nature and character of the networks that Net.Sense is managing.

3.2 Randomized Graph Covers

3.2.1 Background & Theory

A walk is said to cover a graph G if it visits each vertex of G at least once [108, p1]. Walks are useful in discussions where it is desired to process every region in a space. Robot exploration, vaccuuming, terrain exploration, and painting are all applications where some action (taking pictures, running a vacuum, etc) is performed at each point in a space.

Ordinarily, such tasks can be performed quite adequately by looping over some route [102, 103, 105]. However, for some applications (notably, collecting log messages and performing integrity checks) routines and patterns lead to the so-called 'night watchman' problem, where a thief who times the watchman's walk can schedule his theft when authority will be busy elsewhere.

For such applications, the path our 'watchman' takes should be randomized, so that he wanders around without a pattern. This neatly addresses the predictability problem brought on by his cyclic path, but creates two others: efficiency and a lack of predictability. More watchmen are needed because they are exceedingly unlikely to consistently follow an optimal path. The principle of resource parsimony requires using as few watchmen as possible, while still being able to demonstrate that they check all rooms of a building in a timely fashion.

A variety of results have been derived in this area. Among the most useful:

1. The cover time for 1 walker on K_n is $O(n \log n)$ [18, p2].
2. The cover time for 1 walker on any graph is $O(n^3)$ [18, p18-19].
3. Consider a graph G with n vertices and m edges. The expected time for p walkers to cover G $E(C_p)$ is of order $O(\frac{m^2 \log n^3}{p^2})$ [19, p6].

3.2.2 Application

Randomized graph covering is, at present, a topic of academic interest. We are not aware of any significant practical applications using it. However, some groups (including the author) are interested in exploring its utility in networking contexts for virus detection, data collection and mining, and network control [106].

3.2.3 Net.Sense Connection

Though no published applications of randomized graph cover theory have been found, the results given above are useful as load-management and predictive heuristics for the Net.Sense sensor platform population. These relations provide a method to estimate starting values for the Net.Sense agent population and movement rates, based on network size and structure, coverage requirements, etc.

3.3 Probability & Statistics

As described above, Net.Sense trades sensor predictability for enhanced security properties. Here, we discuss the mathematical tools employed to provide quality of service (QoS) guarantees within the Net.Sense system.

It is assumed that the reader possesses basic knowledge of probability and descriptive statistics, particularly probability distributions, population and sample means, variance and standard deviation, and the construction of confidence intervals for random variables associated with populations of known distributions.

3.3.1 Background & Theory

The statistics in Net.Sense bear a rather hefty burden. They are responsible for ensuring that the agent population fulfills the operator's QoS requirements, independent of the circumstances. Because of the unpredictability of the processes driving the managed network (users and their deadlines, media blitzes, and so forth), we feel it is unsafe for Net.Sense to assume any particular underlying distribution for the data it collects. Hence, all measures and evaluations are performed with non-parametric statistics. These techniques have the advantage that they do not rely on a known distribution [55, p75].

Net.Sense employs statistics to assess the state of a particular variable for all elements of a given population (typically, the managed network) and to determine confidence that these variables are maintained within acceptable (operator-defined) limits. In particular, Net.Sense makes extensive use of 1-sided distribution-free tolerance bounds, and their two-sided cousins (called intervals).

A 1-sided distribution-free tolerance bound [55, p91] (hereafter DFTB) describes a region (the $100p$th percentile) of a sample population such that at least that percentage of the population will fall on the desired side of the given bound with confidence $100(1-\alpha)\%$. 2-sided DFTBs (also called DFTIs, for Distribution Free Tolerance Interval [55, p90]) serve the same purpose, but make statements regarding population members falling within an interval rather than merely above or below a certain point [55, p91].

Before DTFBs are discussed with rigor, a brief review of the binomial distribution is required. The equation and subsequent explanation, are described in detail in [55, p78].

$$B(x';n;p) = Pr(x \leq x') = \sum_{i=0}^{x'} \binom{n}{i} p^i (1-p)^{n-i}$$

where

$$\binom{n}{i} = \frac{n!}{i!(n-i)!}$$

In other words, the probability of obtaining no more than x' failing objects in a random sample of n units from a given population is equal to the sum of

the probabilities that many objects will fail independently, each object failing with identical probability p.

It is also worth observing that, in certain situations, $B(x'; n, p)$ can be approximated by $\Phi\left(\frac{x'+0.5-\mu}{\sigma}\right)$, which is the probability that a random variable, distributed normally with mean 0 and $\sigma = 1$, is less than z [55, p79].

Often, we are interested in determining an upper DTFB for a random variable of the population. For example, a Net.Sense operator may wish to make statements like "I am 95% confident that 99% of the managed network will experience interarrival times less than 60 minutes of an agent performing service X". To implement such a statement, Net.Sense must compute a DTFB for X-agent interarrival times. Such an interval looks like

$$EPYP(n, l, n+1, p) = 1 - B(l-1; n, p) \geq 1 - \alpha$$

, where n is the sample size, l is the number of sample points to discard as outliers, p is the target population percentile, and α is the error in our interval. In our example, $p = 0.99$, $\alpha = 0.05$, and $n = 100$ [55, p89].

In order to make these analyses, sufficient data must first be collected. To determine how much data is required for our DFTBs, the following equality for the minimum sample size is used:

$$n = \frac{\log \alpha}{\log p}$$

So, in the example above, we require

$$n = \frac{\log \alpha}{\log p} = \frac{\log 0.05}{\log 0.99} = 298.07$$

Hence, we must take a sample of 299 units.

3.3.2 Application

DFTBs are part of a general statistical toolset, and as such their application is far more general than described here.

3.3.3 Net.Sense Connection

Net.Sense should provide the operator the ability to impose QoS requirements on random variables in the Net.Sense system. Variables that may be monitored and/or controlled include:

1. Interarrival time for agents with given sensor kits.
2. Intensity (rate) of security network events.
3. Process counts, load, swap space in use, or other data from managed nodes.

4. Net.Sense resource consumption (agent move rate, agent size, etc.).

The first of these is most important as it allows the Net.Sense operator to control the intensity of the scrutiny Net.Sense places the system under. Effectively, it allows the operator to control the collection process itself. The second and third items are focussed on the data gleaned from the collection process, and permit the operator to specify the managed network's operating conditions. The last item permits the operator to specify the resource limits under which Net.Sense must operate.

3.4 Review

This chapter has provided background on and relevance to the Net.Sense system of three areas of mathematics: graph theory, distribution-free tolerance bounds, and randomized graph coverings. Graph theory is essential as it provides the language used to formalize managed networks. DFTBs will provide Net.Sense operators the ability to make statistically valid claims about the service afforded by the managed net. Finally, randomized graph cover theorems provide starting points when estimating network configuration values.

Part III

Goals and Hypotheses

Chapter 4

Goals & Hypotheses

4.1 Goals

The authors have been told that the construction of a fully operational Net.Sense system could become a life long project. The objective of this thesis is to begin clearing the road toward the fruition of such a system. This objective was broken down into four goals, as follows:

1. To provide a detailed decription of the Net.Sense vision,
2. To implement a framework that validates and explores that description, as described in the development plan,
3. To validate and test that framework by implementing a simple Net.Sense system, as described in the development plan,
4. To conduct experiments and explorations with the system, as described later in this chapter.

4.1.1 Vision Description

Details of the Net.Sense vision have been discussed piecemeal throughout the literature review, as doing so helped validate the reviewed topics. However, to this point no clear, unifying vision has been presented. This section rectifies that omission by providing high-level, unified views of the Net.Sense vision from organizational and operational perspectives.

4.1.2 Framework Prototype

A prototype development framework was constructed to validate and explore the nuances of the Net.Sense vision. This prototype provides proof of concept implementations of Net.Sense support functions such as Tools and the InfoSpace.

With this framework, working (albeit limited) Net.Sense systems can be built and experimented with.

The prototype framework is based on several publically available software packages, notably the JADE agent framework and JavaSpaces. JADE [12], developed by Telecom Lab Italia, provides classes for developing FIPA-compliant [3] mobile agents. JavaSpaces [47], provided by Sun Microsystems, builds off Sun's RMI and Jini code to provide a distributed, Linda-like [93] distributed space.

With these tools in hand, the prototype framework can be developed. Presently, this includes the base-agent extensions, modular Tool and Behavior systems, distributed singletons and data structures, as well as extensions to the JADE behaviour system to more readily facilitate subsequent development. Each of these components are described in detail in Section 6

4.1.3 Test System

The test system serves two purposes: to validate design choices made in the prototype framework, as well as to provide a vehicle for exploration of its operational characteristics. The test system builds off the prototype framework to demonstrate a simple Net.Sense system that consists only of a sensor and reasoner community. (The justification for this deficiency is expanded on in the future work section (9.2) and testbed system chapter (7).)

As implemented, the test system collects information about processes running on a managed network of Solaris hosts. This information is stored in the JavaSpace implementation of the InfoSpace provided by the prototype framework, where it is processed by reasoner agents. These agents use the raw sensory data to compute metadata and statistics to be used in testing our hypotheses.

4.2 Hypotheses & Explorations

As previously indicated, the test system serves two related purposes. First, it serves to enables the exploration of the consequences of the design choices made in the prototype framework. Second, it provides a testbed for collecting data to validate our hypotheses. In this section, our hypotheses and the rationale behind them are discussed.

4.2.1 Hypothesis 1: Scalability

The primary experimental hypothesis states that to maintain a given quality of service in the managed network, the number of agents required will scale linearly with the number of hosts in the managed network. More formally, given:

- n, the number of hosts in the managed network,
- a, the number of deployed sensory agents,

- t, the mean time an agent spends at a host, and
- Q, the QoS provided to the managed network

we hypothesize that these factors obey a relation of the form

$$aQ = nt$$

To this point, the term "QoS" has been used rather loosely, as its meaning varies with the particular application. However, such informality does not suffice for a statement of hypothesis. Therefore, for the purposes of Hypothesis 1, define the QoS delivered to a given managed network to be the 90th percentile distribution free tolerance bound (DFB) of the collection of managed hosts' mean Inter-Arrival Times (IAT)s, over time. This is explained in more detail in Section 7.2.3, but can be summarized as follows:

Given a collection of hosts' mean times between agent visits $\overline{IAT}\chi$, the QoS provided to the managed network is defined to be the duration Q such that, with 95% confidence, 90% of the $\overline{IAT}_\chi$ will be less than Q.

Justification

Consider a managed network with n hosts and a sensor agents, connected in a flat K_n topology. Assume that agents move from host to host once every t seconds. Then, if the required QoS is Q seconds, the system must visit, on average, $\frac{n}{Q}$ hosts per second. Knowing that a agents visit a total of $\frac{a}{t}$ hosts per second, we see that $a = \frac{nt}{Q}$ agents would be needed were the agents to collectively follow an optimal set of paths through the network. An optimal path is a path in which no agent visits a node that has already been (or is currently being), visited.

Informal experimentation with an idealized model using Brookings Institute's Ascape [1] modelling system provided initial validation for Hypothesis 1, and suggested that roughly one agent per four managed hosts was required for a given quality of service. This experimentation also supported the common-sense idea that swarm techniques are in fact effective for load-balancing and network coverage. Further research (see Sections 3.3) indicated that one agent would cover our K_n network topology in $O(n \log n)$ steps.

4.2.2 Hypothesis 2: Swarm Control

Techniques borrowed from swarm intelligence and biological population models will effectively control and balance the overhead incurred by the sensor community. This hypothesis is considered in two parts: controlling the overhead, and balancing this overhead.

Part 1: Overhead Balance

This hypothesis is qualitative, and simply states that the sensor swarm will not unduly load any particular host or hosts.

There are many factors that affect the swarm's distribution across the network. Movement restrictions or preferences as well as control in any of its incarnations (tropisms, direct command, sheer random chance) all play a role in determining where the swarm collectively 'is' at any moment in time. As a result, care must be taken to specify which of these factors are in play in the system under test.

For the purposes of Hypothesis 2.1, the system under test is considered not to implement tropisms, nor to respond to directed movement commands. As a result, the swarm distribution is subject only to random chance. Further, the agent movement code is considered to treat the entire network as directly connected, with no movement penalties. Hence, the managed network (from the swarm's perspective) is of the K_n form.

Justification

Intuitively, a group of a agents moving randomly on K_n is equivalent to randomly choosing a of n elements, with replacement. This can be seen by considering the network one agent at a time. Because sensor agents neither keep a travelog nor know where other sensor agents are, no consideration is given to these factors when selecting a destination. Hence, each agent may be considered as though it were the only one on the network. The 'wander' code used in the test system's swarm is written to select a host at random (including the host it currently occupies). As a result, the probability that any given host χ will be occupied by this agent is $\frac{1}{n}$. Thus, the odds that c agents will simultaneously occupy any given host χ is approximately $\frac{1}{n^c}$.

Part 2: Population Controls

Hypothesis 2.2 states that a Net.Sense system will be able to provide assurances that it meets QoS requirements while minimizing the resources incurred by the sensor community.

Justification

Process control is a relatively well-understood area. Because the swarm population can be treated as a process variable, it can be controlled in the usual way, provided suitable actuators are in place for the control system to create or destroy swarm agents as needed.

Hypothesis 2.2 should be trivially confirmable by implementing, within Net.Sense, a process control system to manage the swarm population. This implementation is seen as having two parts: the process control algorithm itself, and the sensor/actuator systems that enable its sense-affect cycle. The first of these two parts is well understood and, we believe, reduces to an implementation issue.

The second part, being the sensors and actuators for the control algorithm, are somewhat less well understood. While the results of the present develop-

ment effort do include a primitive system which augmented the swarm as needed, careful consideration is required when reducing the swarm's size. Care must be taken to shut agents down 'gently', so as not to lose data structures or singletons or create inconsistent configurations on host systems. While conceptual solutions to these issues exist, they have not yet been put into practice. Once such a system is implemented, further research into response times, predictive or anticipatory modifications, etc. is in order.

Part IV

Design and Implementation

Chapter 5

High-Level Design

Details of the Net.Sense vision have been discussed piecemeal throughout the literature review, as doing so helped validate the areas researched. However, to this point no clear, unifying vision has been presented. This section rectifies that omission by providing, in general terms, a high-level Net.Sense vision. The core of this vision is a self-introspecting, self-managing process that senses, reasons about, and affects the network that is its environment. When such a process is integrated into the managed network, the network effectively manages itself.

5.1 Organizational Perspective

This section provides an overview of the functional components of a Net.Sense system: the agent community, the InfoSpace, and the managed network.

5.1.1 Agent Communities

Agents are the eyes, mind, and hands of a Net.Sense system. The agents exist on hosts in the managed network, providing services to each other and the system's users (who are likely to be system administrators). The agents are divided into three logical communities: the sensor community, also called the swarm; the reasoner community, and the actor community.

The agents of the swarm are Net.Sense's eyes. They wander randomly around the managed network, collecting data from managed hosts and publishing it into the InfoSpace. These agents are largely oblivious to each other, and thus avoid incurring large network and computational overhead. In advanced Net.Sense systems, agents in the swarm can be retasked by agents in the reasoner community.

The agents of the reasoner community collectively form Net.Sense's brain. It is the command and control center, responsible for processing incoming data, deriving courses of action, and tasking agents in the actor community to carry out its directives. The agent community contains librarians, statisticians, do-

main experts, task planners, community maintenance agents, and user interface agents. Agents in this community are driven either independently or by the arrival of new data in the InfoSpace. The reasoner community is responible for analyzing the state of the managed network, assessing any potential issues, and formulating action plans as needed.

The agents of the actor community are Net.Sense's hands, responsible for implementing all desired changes in the managed network. Agents in this community are tasked by reasoner agents to take specific tools to selected hosts and apply them according to a precomputed script. The actor community is responsible for efficiently, securely, and accurately changing the state of the managed network.

Together, these three communities implement a sense-affect cycle within the managed network.

5.1.2 The Sense-Affect Cycle

It is felt that to some extent, all living organisms interact with their environment in a cyclic fashion. The organism first experiences or senses its environment, then assimilates, and finally takes action on that sensation. This 'sense-affect' cycle forms the basis for Net.Sense's interaction with the managed network.

The sense-affect cycle consists of three phases: Sense, Reason, and Affect. (See Figure 5.1) The Sense phase is continuous, performed by the agents of the sensor community. The Reason phase is keyed from the results of the Sense phase. In this phase, reasoner agents examine data within the InfoSpace and assess its relevance to the programmed objectives of the Net.Sense system. Should abnormal conditions be detected, the reasoner community uses its knowledge-base and specific, tailored system policies to identify root issues, select workable solutions, develop courses of action, and finally task actor agents to carry them out. The Affect phase is performed by the actor community and is driven by commands issued by Reasoner agents. In this phase, the courses of action developed by the Reasoner community are implemented within the managed network.

The state changes brought about by the actor community should be visible to the swarm. This creates a feedback loop that enables Net.Sense systems to evaluate their operational effectiveness. Net.Sense systems determining that they are unable to resolve problems could try alternative methods or tools, potentially notify operators, request better tools, etc.

From a philosophical perspective, the Net.Sense system is analogous to the 'ghost' in the machine. In its ultimate form, a Net.Sense system is an agent-like construct 'living' *within* the managed network. The agents of the sensor community are its eyes and ears; the reasoner community its brain; the effector or actor community its hands. In many ways, Net.Sense resembles a semiautonomous robot operating in a virtual space.

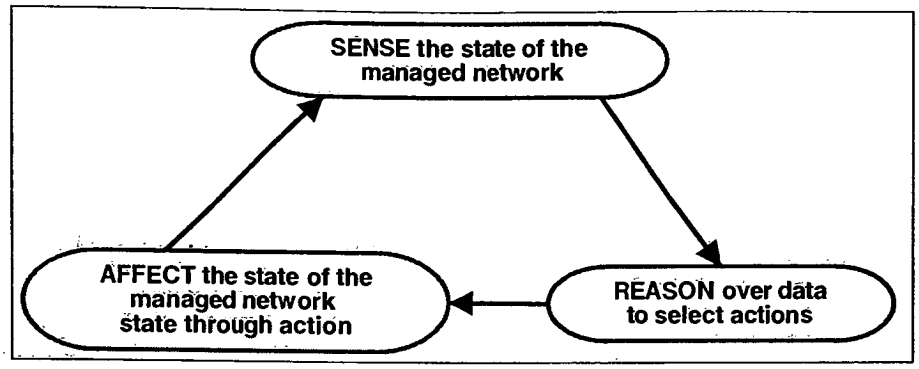


Figure 5.1: The Sense-Affect Cycle

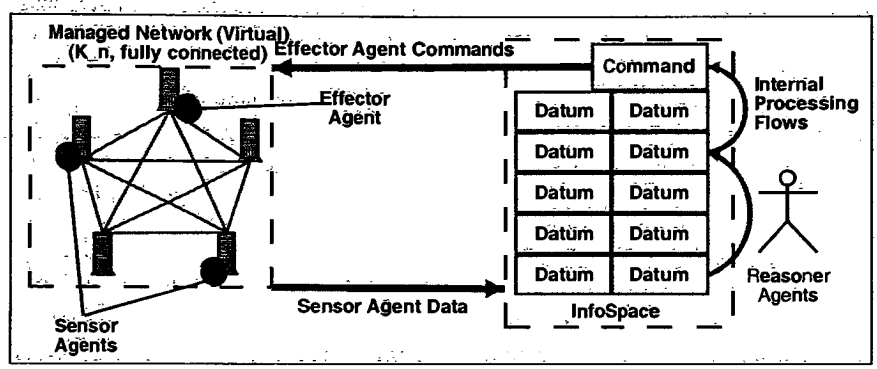


Figure 5.2: Data Flows and the InfoSpace

5.1.3 The InfoSpace

The Information Space, (more commonly, 'InfoSpace'), is the conceptual repository for all nontrivial information in a Net.Sense system. This includes raw sensory data, computed meta-data such as interarrival times, statistics, derived system conditions, pending tasks, and agent state. The Net.Sense vision for the InfoSpace is akin to a temporally enhanced distributed database. It is to be an efficient, secure, globally accessible, object oriented data store that serves agents as command channel, collaboration medium, and data warehouse.

As the author has learned, the key to understanding an information system is understanding the location and flows of its information. A fully operational Net.Sense system has three logical data flows, one for each of the three states of the sense-affect cycle. (See Figure 5.2) The first flows from the sensor community into the InfoSpace. The sensor agents are dispersed throughout the managed network, but each is virtually 'local' to the InfoSpace by virtue of a connector.

The second logical data flow is internal to the InfoSpace. It consists of two

subflows: one feeding into the reasoner community for processing, one feeding results back into the InfoSpace. Some of these results, such as statistics and other metadata, remain in the InfoSpace. Other results, such as commands to agents, form the third virtual flow that leads from the reasoner community out to the actor community living in the managed network.

It should be noted that these flows are strictly logical because the InfoSpace is virtually 'local' to all agents in the Net.Sense system. InfoSpace implementations may provide their functionality in different ways, and may elect to create real, concrete data flows that differ from the logical model.

5.1.4 Agent Structure

Net.Sense agents are envisioned as dynamically reconfigurable, modular, collaborative agents. They are composed of three components: a chassis, a tool collection, and a behavior collection. The chassis is the immutable framework around which agents are constructed. Onto this framework tools and behaviors are loaded, a set of which is collectively called a 'load-out'.

Tools provide raw agent capabilities. These capabilities may enable the agent to interface with (sense or affect) the managed system, or to perform tasks such as reasoning or other data-centric computations. Sensor/affector tools might give an agent the ability to determine which processes are running on the local host or perhaps to reconfigure a network interface.

Behaviors provide the 'knowledge' that enables an agent to use tools to provide services to itself and the outside world. Behaviors determine under what conditions a given tool will be used, and therefore they are a part of the agent's autonomy layer. For example, a behavior responsible for creating a new file system could require its agent to remain on its current host for the duration of the operation.

While only behaviors should invoke tools, it is not the case that there must be exactly one tool per behavior, or even that a given behavior is tied to a particular tool. Rather, behaviors providing complex services may require a group of related tools. With semantic tagging (to be discussed shortly), we will see that behaviors need not always specify which particular tools they need, so long as they accomplish their assigned tasks.

The service provision process is outlined in Figure 5.3. As the figure illustrates, agents receive service requests that the agent chassis (specifically, its communication center) routes to an appropriate behavior or behaviors. The behavior assesses the request and decides whether or not to honor it. (Hence, much of an agent's autonomy is found in its behaviors.) If the agent elects to act on the request, it does so by configuring and engaging the tools required to provide the service. As stated earlier, the tools provide the the abilities, while the behaviors embody the knowledge of how and when to use them. At present, the question remains open as to whether to separate the 'how' knowledge from the 'when' knowledge.

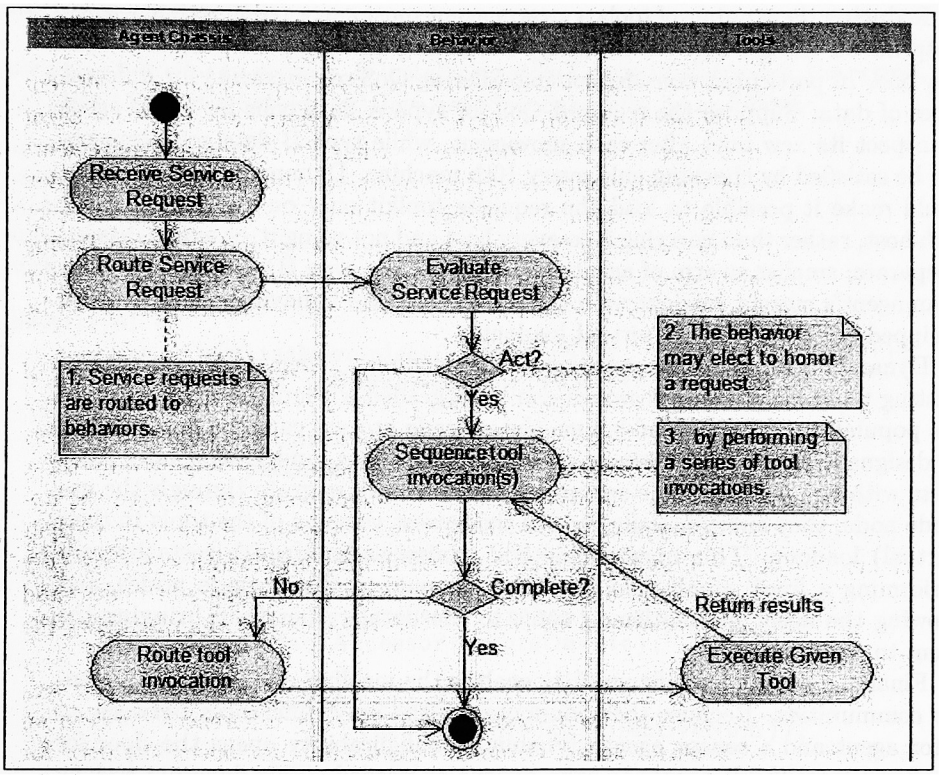


Figure 5.3: Service Provision in the Net.Sense Vision

5.2 Functional Perspective

This section discusses the advanced functional aspects of Net.Sense systems.

5.2.1 Dynamic Reconfigurability

In the Net.Sense vision, agents' may change their loadouts while they are live in the managed network. This ability provides several benefits, but the most interesting of these are footprint reduction, live upgrades, and the melding of the sensor and actor communities.

Reducing the number of tools a swarm agent must carry reduces that agent's footprint on the managed network by simulating many agents where there is only one. In particular, consider setting differing QoS requirements for different types of data. Suppose the required QoS for type A data calls for every agent to inspect its new host after each movement, while type B data requirements can be satisfied by checking only every 10th machine. Dynamic reconfiguration would make it possible to load the requisite toolkit only on arriving at every 10th host, rather than carrying it to every host and not using it, or worse, having a separate, smaller swarm of agents bearing only the type B sensor. The union of an agent's current loadout with the set of tools and behaviors that it can be equipped with is called its 'virtual loadout'.

Dynamic reconfiguration enables live upgrading. When newer versions of existing tools are ready, the swarm can be upgraded without 'recycling' its entire population. As an added bonus, the upgrade process can be made possible by designating managed hosts as swarm update nodes, and issuing only these hosts with the new tool. As swarm agents randomly migrate through the upgrade node, they automatically receive the newest versions of the tools in their (virtual) loadout. This upgrade method minimizes bandwidth costs by only performing a fixed, small number of deployments to the upgrade nodes, and allowing the agent population to come to the data, rather than bringing the agent to the data.

Finally, dynamic reconfigurability enables the melding of the actor and sensor communities. Because their loadouts are the only differences between an actor agent and a sensor agent, dynamic reconfigurability opens the door for 'drafting' local 'field' agents to take actions as needed. This dynamic tasking further reduces the overhead required to service the managed network, as the actor community becomes wholly virtual, manifesting only when needed in the form of an addition to a given field agent's virtual loadout.

5.2.2 Goal-seeking and Semantic Tagging

Rudimentary Net.Sense implementations will likely require the system administrator to manually script higher-level tasks from basic tool actions. However, in a fully-realized Net.Sense system, goal-seeking AI combined with a significant fact base will enable the system administrator to specify the end result and

invariants. For example, the user directive might be "ensure that all non-server hosts have patch X, but do not impact any user sessions."

The reasoner community, on receiving this directive, might reference 'impact' and 'user session', determining that a given set S of actions ('reboot', 'kill process', etc) will affect user sessions, and establish those actions as forbidden. Given the start state (a potentially unpatched system), the desired end state (a patched system), and a list of actions not to take, the reasoners can attempt to find a series of agent-executable steps (tool executions, etc) that will bring a given system from the start state to the goal state without performing any of the forbidden actions.

This script would be formatted in a fashion suitable for an actor agent's scripting behavior to process. If installing patch X were nonintrusive, the script (rendered here in a hybrid Perl-Java pseudocode) might be as follows:

```
use patchCheck;
use patchAdd;
use whoSensor;
use sysReboot;
use ISpaceAccessor;

onArrival {
        tc = new taskComplete(directiveNum, getCurrentHost());
        wander() if (ISpaceAccessor.read(tc));

        if (!patchCheck('patchX')) {
                patchAdd(patchID => 'patchX');
        }

        if (whoSensor.getCount() == 0) {
                sysReboot.execute
                        (mode => 'reboot', delay => '120')
                ISpaceAccessor.ensureOne(tc);
        } else {
                ISpaceAccessor.ensureOne
                        (new rebootFlag(directiveNum));
        }
        wander();
}
```

Paraphrasing, this script would ensure the executing agent's loadout included the patchCheck, patchAdd, whoSensor, sysReboot, and ISpaceAccessor tools. On arrival at a given host, the agent would determine if this host had already been patched. If so, the agent would continue wandering. If not, the host would be patched, and checked to determine whether it was presently safe to reboot. If so, the reboot would be scheduled and a completion notification written into the JavaSpace. If not, a flag would be written into the InfoSpace for later reboot attempts. Finally, the agent would migrate off the host.

Note that the script is not as efficient as it could be. In particular, this script would cause hosts with long-running user sessions to be repeatedly checked for

the patch. We envision that as script-generating rule sets evolve, proposed scripts would be checked for significant performance issues, or generated with progressively more efficient heuristics, for example, by setting and checking for a 'patched' flag in the InfoSpace before engaging the patchCheck tool.

5.3 Review

This chapter has promulgated the Net.Sense vision at a high-level. Fundamentals including agent community organization and function, the sense-affect cycle, and the InfoSpace were addressed. In addition, aspects of more advanced Net.Sense systems such as semantic tagging, automated actor scripting, and dynamic reconfigurability have been discussed.

Chapter 6

Prototype Framework

This chapter provides a mid-level design view of the framework as implemented, and indicates which portions of the high-level vision are realized by each component.

Before proceeding, it is important to understand the distinction between the terms 'behaviour' and 'behavior' or 'NSBehavior', which are used throughout the remainder of this thesis. An 'NSBehavior' (NSB), or simply 'behavior', is a subclass of the `ns.infra.NSBehavior` class that implements the 'behavior' concept used in the vision statement in Section 1.3.1. 'Behaviours', with a 'u', are classes implementing the tasks and subtasks that behavior code is decomposed into for the benefit of the JADE scheduling system. No conceptual analog is present in the Net.Sense vision as behaviours are a consequence of specific implementation decisions and are therefore hidden at the vision's level of abstraction.

6.1 Overview

The prototype Net.Sense framework consists of three primary packages: `ns`, `stat`, and `jade.core`.

`ns`, contains the present implementation of the Net.Sense framework prototype.

`stat`, contains the bulk of the code for the testbed system. While the testbed is distinct from the framework, several of the `stat` package's classes are found in this package. This is the case because during the development process, several of the original `stat` classes were generalized to a degree that merited their migration into the main `ns` hierarchy. As continuity concerns precluded their migration during this development cycle, they remain in the `stat` package. These displaced classes are discussed in this chapter, while the remainder of the testbed system is discussed in Chapter 7.

`jade.core`, contains modified versions of certain JADE classes, in addition to those Net.Sense extensions made vastly simpler by their inclusion in this package's scope.

6.1.1 Package ns

The ns package consists of 5 subpackages:

`behaviours`, contains behaviours that modularize and simplify NSBehavior client code.

`dam`, contains the 'distributed array metadata' (DAM) family of distributed data structures (DDS).

`infra`, contains the core Net.Sense infrastructure.

`onto`, contains rudimentary Net.Sense ontologies.

`tool`, contains the tool implementations and their supporting classes.

Package `ns.behaviours`

The `ns.behaviours` package is a collection of helper behaviours that fall into two categories: those facilitating DDS creation and manipulation, and those of general architectural utility.

Package `ns.infra`

The `ns.infra` package contains the majority of the Net.Sense infrastructure. This includes the base Net.Sense Agent architecture, the Net.Sense Behavior and Tool frameworks, and portions of the InfoSpace implementation.

Package `ns.dam`

The `ns.dam` package contains classes implementing a family of distributed data structures (DDS) including Distributed Array Metadata (DAM)s, Distributed Set Metadata (DSM)s, and Singletons. The package was named for its primary class hierarchy, the DAM hierarchy, which was named for its original array-like API. The package's contents have evolved considerably since the package was named. For example, the DAM API has become more list-like, and both DSMs and Singletons were added to the package. As a result, the name 'dam' is somewhat confusing. The name is retained for continuity and is likely to change in a future release.

Package ns.onto

This package contains ontologies created while exploring the JADE messaging system. The original intent was to implement rudimentary semantic tagging (see Section 5.2.2) for InfoSpace data by marrying data objects with ontologies that described them. The design rationale was that self-descriptive data could be more easily reasoned over by client communities of diverse-source expert systems. Owing to time constraints, the existing ontologies presently serve only to assist in the automatic encoding of certain intra-agent messages.

Package ns.tool

This package contains a selection of tools that would be part of an agent's load-out. The most highly developed of these is the ProcessLister, which has been modified to work with the preliminary InfoSpace implementation. The remaining tools in this package, while notionally functional, have not been modified and are not useful in test systems at the present time.

6.1.2 Package stat

The bulk of this package's contents are related to the testbed system described in Chapter 7. However, as noted above, certain classes in this package have been generalized sufficiently to merit inclusion in the Net.Sense framework, to which they will formally migrate in the next release.

6.2 Expansion: ns.behaviours

As noted above, the behaviours in this package fall into two categories: those that facilitate DDS creation and manipulation, and those that are of general architectural utility. Each of these categories is discussed in turn.

6.2.1 DDS-Oriented Behaviours

These classes facilitate creating and manipulating DDSs (a process discussed in more detail in Section 6.5.3). The major classes in this category are organized as shown in Figure 6.1. These behaviours greatly reduce the programmer effort required to manipulate DDS in JADE's cooperative scheduling environment.

GenericOpBehaviour

The Generic Op Behaviour is a metabehaviour that captures the generic process for acquiring and subsequently using a data element in the JavaSpace. The process has five phases: acquisition, lock, user, unlock, and cooldown. (See Figure 6.2) The premise underlying the GenericOpBehavior is that high-level process flow remains identical regardless of the data structure the client wishes to manipulate or the type of manipulation performed. Each phase can itself be

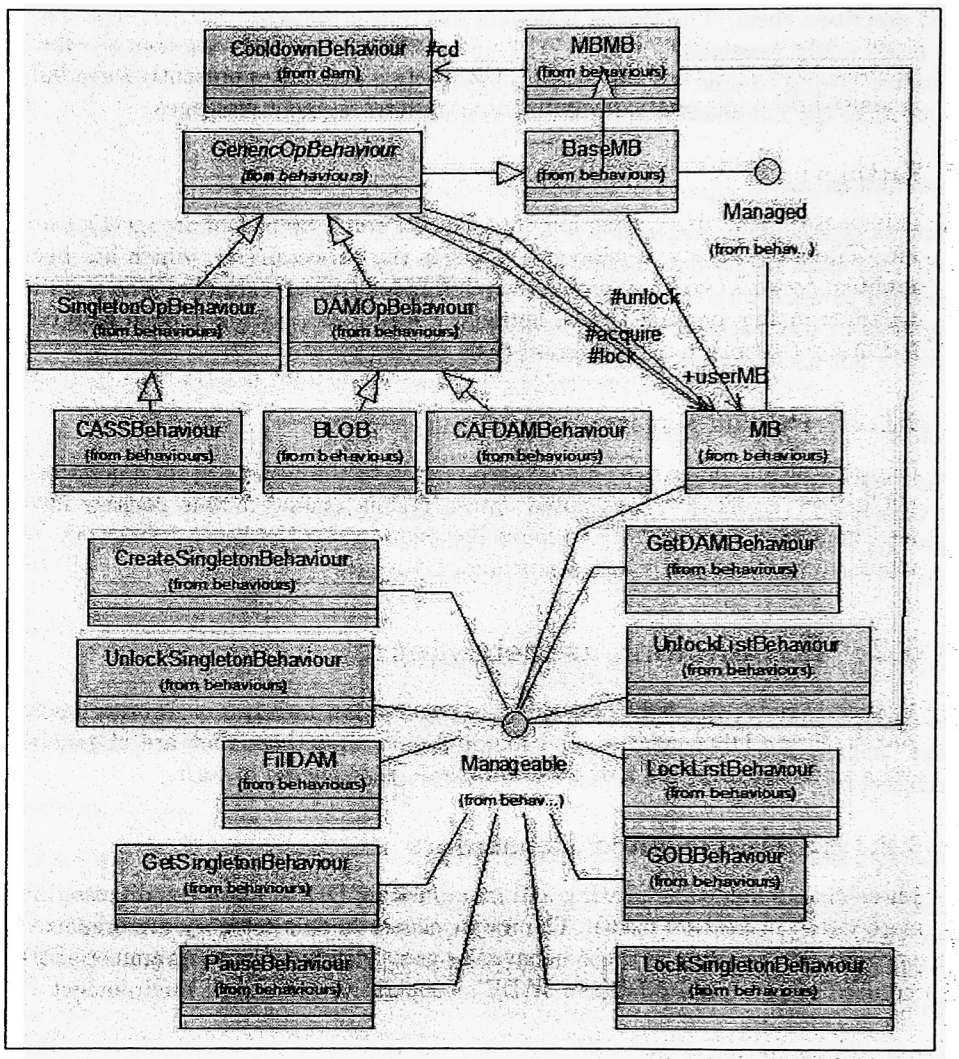


Figure 6.1: The DDS Behaviours

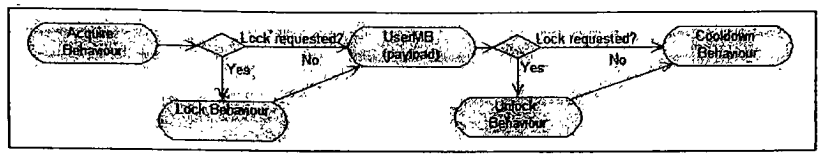


Figure 6.2: The GenericOpBehaviour Control Flow

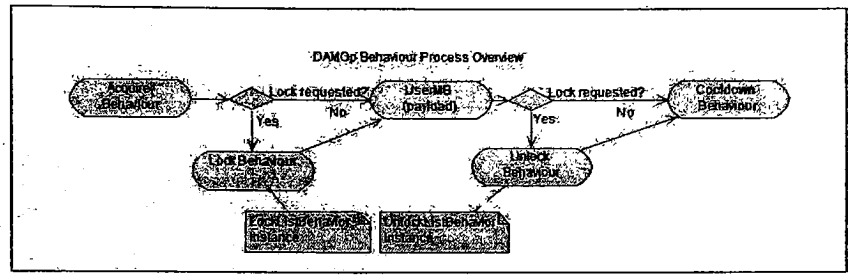


Figure 6.3: The DAMOpBehaviour Control Flow

a nontrivial process, as it is realized by the MB class described in Section 6.4.1.

The `GenericOpBehaviour` class is typically used by overriding its constructor and assigning the desired behaviors to its various phase variables. These phase variables are functionally similar to function pointers in that they allow the GenericOpBehaviour's functionality to be changed at run time. As an example, the `DAMOpBehavior` class overrides GenericOpBehavior's constructor to schedule Lock- and UnlockListBehaviours for the GenericOpBehavior's lock and unlock phases (see Figure 6.3).

DAMOpBehaviour

The DAM Op Behaviour defaults the GenericOpBehaviour's locking phases to Lock- and UnlockListBehaviour objects. See Figure 6.3.

SingletonOpBehaviour

The Singleton Op Behaviour defaults the GenericOpBehaviour's locking phases to Lock- and UnlockSingletonBehaviors in much the same manner as the DAMOpBehaviour. (See Figure 6.4)

GetSingletonBehaviour

The Get Singleton Behaviour reads a particular Singleton, if it exists, from the JavaSpace. The behaviour accomplishes this through a SingletonConnector that

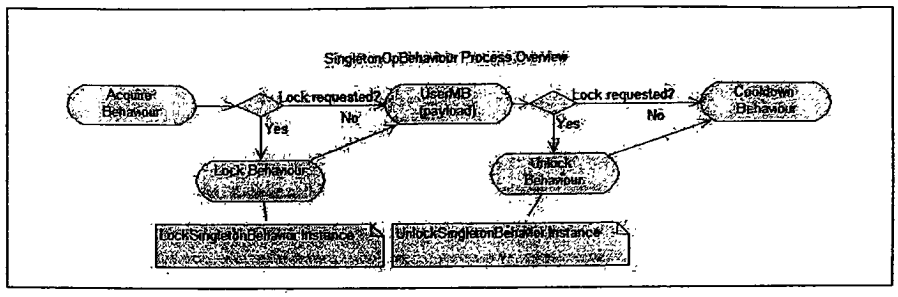


Figure 6.4: The SingletonOpBehaviour Control Flow

performs the actual interaction with the JavaSpace. While not overly complicated, this code has been separated out into its own behavior for scheduling purposes, as it may entail significant delays.

(Un)LockSingletonBehaviour

These behaviours take the necessary steps to lock (take from the JavaSpace) and return (put into the JavaSpace) a particular Singleton, if it exists. These actions are accomplished through the SingletonConnector, which performs the actual interaction with the JavaSpace. While not overly complicated, this code has been separated out into its own behavior for scheduling purposes, as it may entail significant delays.

CreateSingletonBehaviour

This behaviour takes the necessary steps to create a new Singleton with the specified parameters. The behaviour is primarily intended to be called from a CASSBehavior that initializes the Singleton in addition to creating it. The CreateSingletonBehaviour (through the SingletonConnector and SingletonFactory) ensures that duplicate Singletons are not created.

CASSBehaviour

The Create and Set Singleton Behaviour, or CASSBehaviour, is a SingletonOpBehaviour that overrides the acquisition stage in order to create the specified Singleton if it does not already exist. Note that this does *not* guarantee that the contained userMB has access to the specified Singleton. Should the requested Singleton already exist, but have been locked by another client, the Singleton will be unavailable in the JavaSpace, will not be (re)created by the SingletonFactory, and hence will be undelivered to the userMB.

(Un)LockListBehaviour

These behaviours remove and return the specified DAM from or to the JavaSpace by means of the DAMConnector class. While not overly complicated, code for these actions has been separated out into their own behaviors for scheduling purposes, as they may entail significant delays.

CAFDAMBehaviour

The Create and Fill DAM Behaviour (CAFDAMBehaviour) acquires the specified DAM (constructing it if it does not exist), locks it, adds to it the elements in the given list, and unlocks it again. This effectively translates a Java List into a DAM in the JavaSpace. Note that this process does *not* recurse. Elements enumerated from the given Java list will be added to the acquired DAM and thence to the JavaSpace. Users wishing to translate existing nested data structures into equivalently structured DAMs must presently do so manually. The DAMs must be created manually, innermost-nested list first, and then sequentially added to their respective containing DAMs. See the `ns.tool.ProcessLister` for an example of this process in action.

GetDAMBehaviour

This behaviour attempts to read the specified DAM from the JavaSpace via a DAMConnector. While not overly complex, this operation can entail significant delay and is therefore separated into its own behaviour.

GOBBehaviour

This behaviour attempts to read the specified DAM from the JavaSpace via a DAMConnector, creating it if necessary. While not overly complex, this operation can be time consuming, and it is therefore separated into its own behaviour. Should the DAM already exist, but have been locked by another client, the DAM is unavailable in the JavaSpace, is not (re)created by the DAMFactory, and hence is undeliverable to the userMB.

BLOB

The Basic List Operation Behaviour (BLOB) is a DAMOpBehaviour that overrides the GenericOpBehavior's acquisition phase with a GetDAMBehaviour. This has the effect of acquiring (but not creating) the specified DAM before passing control to the given user behaviour.

6.2.2 Architectural Behaviours

While often instrumental to DDS creation and manipulation, these behaviours are sufficiently general to be applicable outside that context.

CooldownBehaviour

The Cooldown Behaviour performs no task, but provides a fixed exit state for all complex behaviours. This simplifies the generalization of behaviour classes.

PauseBehaviour

The Pause Behaviour delays the containing behaviour for a given length of time, plus or minus a small (max 128ms) variation. (The variation helps ensure that race conditions are resolved.) It does *not* block other scheduled behaviours from executing unless the PauseBehaviour is at some level contained by an MBMB in run-through mode. In that case, the PauseBehavior spins the executing thread.

MBMB

The Managed Behaviour Managing Behaviour (MBMB) provides exception handling while executing subbehaviours, augmented scheduler functionality, simplified subbehaviour registration, scoping (pad) facilities, and other enhancements to the standard JADE behaviour hierarchy. See the MBMB expansion (Section 6.4) for further details.

BaseMB

The Base Managed Behaviour or `BaseMB` class is an error-handling wrapper for user-supplied MB objects. It provides a convenient way to complement an existing managed behaviour's delay-and-retry mechanism with the exception-handling semantics of `MBMBs`.

6.3 NS Behavior System

The Net.Sense vision calls for agent behaviors to provide services to the agent community by mediating access to abilities provided by agent tools. This vision did not address implementation issues, many of which turned out to be non-trivial. Examples include the need for either reentrant tools or a tool activation control mechanism to prevent threads with distinct parameter sets from operating concurrently within tool code. In the prototype, such issues only arise in the IAT and DStat agent classes. The issue is mooted by writing the controlling behaviors to serialize tool use. This does not afford real protection against a malicious user, but it suffices for the needs of the current prototype.

Some Net.Sense behaviors (NSBs) or tools may require an agent to maintain a certain state while they operate. For example, agents ought not move while executing system commands on a host. The prototype agent base therefore includes an NSBMaster (see Section 6.6.1) that permits visibility into whether NSBs are active. It is felt that, in the future, semantic tagging will permit NSBs to state the semantic invariants they require during operation. It is felt that

providing these semantic tags to a lightweight production system embedded in the agent base could greatly simplify the NSB/tool development process.

'Active' NSBs have been discussed, but what does it mean for an NSB to be active? To crystallize this concept, consider the following example of the tool activation process.

The tool activation process begins with the receipt of an ACLMessage by the agent's Comm Center. The `infra.CommCtr` class provides facilities to send and automatically receive and distribute messages. The received message is relayed to all registered NSBs (see Section 6.6.2). Each NSB contains an inner message handler class that is responsible for filtering out irrelevant or inoperable messages. In the present implementation, this filtering consists of determining that the message is 'written' in the correct ontology and contains a valid Poll object. The Poll object is extracted from the message and passed to a poll request handler.

At this point, a valid Poll action has reached the appropriate NSB's handler function. For the purposes of this example, it is assumed that the NSB in question is a Simple Tool Invoker Behavior (STIB) as implemented in `ns.infra.STInvokerBehavior`. The STIB extracts the target tool's name, selects the tool from the agent's toolbelt, configures it with the NSB's parameter set, and activates the tool.

This process differs from the Net.Sense vision in two important ways. First, Poll objects should contain (public) service names rather than the names of the tools loaded into the agent. Tool specifics, like helper functions, should be irrelevant to and therefore hidden from the clients (in this case, the agents). Rather than use tool instance names, agents should request that a specific, uniformly and publically named service be performed. Second, the Poll object should contain any configuration parameters required to perform the service. The NSB should be responsible for validating these parameters and performing other security and autonomy tasks before activating the tool or tools required to perform the requested service. Such an arrangement would also facilitate tool reentrancy, as discussed at the beginning of this section.

The implementor's original intent was for Net.Sense to utilize threading far beyond the single JADE thread per agent. Many factors played a role in the decision not to pursue this effort, including thread (non)serializability, JADE compatibility concerns, and the lack of accepted means to selectively terminate running threads. As a result, significant effort was expended to manually subdivide code into behaviours (note the 'u'!) suitable for the JADE scheduler. These behaviours are analogous to functions in a programming language: they return values, have available (limited) scoping mechanisms, may 'call' other behaviours, etc.

In retrospect, it is not clear that this subdivision was the right decision. Significant complexity is added to the Net.Sense model for the sake of a highly specialized class of behaviors; namely, those behaviors that cannot be threaded by the nature of their interactions with the JADE framework.

A necessary consequence of the present design is that NSB code loses track of the activated tools' state. Generally, a tool's task will be too time-consuming

for execution in its primary entry point. As such, entry point code serves as a task loader that creates and schedules instances of the behaviours containing the main task code. Because this entry point routine must return to permit the newly-tasked behaviours to be scheduled, neither the NSB nor the code in the tool entry point have detailed knowledge of the tool's execution state.

This gulf between JADE behaviours and the Net.Sense code that uses them is bridged by a system of callbacks. On activation, NSBs and tools set an internal flag indicating that they are active. When their entry points schedule the behaviours that actually perform their task, a cleanup callback behaviour called a notification behaviour (NOB) is also scheduled. The NOB is responsible for performing any cleanup actions to run after the primary behaviour sequence finishes. When the last primary behavior completes, the NOB is executed. It changes flags, records error state, and performs other cleanup functions as defined by the caller. By ensuring that the last flag (un)set is that indicating (in)activity, inquiries regarding the activity of the tool or NSB in question produce the correct response.

To summarize, an NSB is 'active' from the moment it begins executing until the last NOB queued by the NSB or its subordinate behavio[u]rs executes and terminates. NOBs serve as a rudimentary asynchronous messaging system that provides the reentrancy guards needed in the prototype framework.

The decision to decompose agent code for collaborative scheduling had significant consequences. These consequences are most visible in the collection of common, generalized, highly granular subtasks that were factored into separate utility behaviour classes. Many of these classes have been placed in the `ns.behaviours` package. They are divided into three groups: DAM/List, Singleton, and Architecture.

6.4 Extensions to JADE

JADE provides a flexible set of Behaviour classes that provide users the ability to orchestrate complex behavior patterns. Chief among these are the SequentialBehaviour, the ParallelBehaviour, and the Finite State Machine (FSM) Behaviour. While the ability to compose FSMBehaviours is quite powerful, the author found its error-handling mechanism somewhat lacking. The JADE system uses numerical result codes, rather than exceptions, to determine the exit state of a Behaviour. While workable for smaller behaviour patterns, this mechanism is inadequate for systems needing to manage failure throughout a complex, multilayer behaviour hierarchy. In addition, it seems to discard the work invested into Java's exception mechanism and its signficant fruits of failure transparency.

Extensions to the JADE Behaviour system were developed for this reason. The extensions consist of modifications to the JADE CompositeBehaviour and Agent classes, as well as new classes and interfaces. The new material consists of the Managed interface and Manageable and Managed Behaviour(MB) classes. The modifications to the existing JADE classes exposed key functional-

ity within the to the new classes, enabling them to provide exception handling and enhanced manageability within the agent. The following discussion focuses on the Net.Sense extensions rather than the modifications to the JADE system.

The cornerstones of the Net.Sense extensions are the MB and MBMB classes. The Managed Behaviour (MB) class decorates CompositeBehaviour with functionality useful in managing complex behaviours. The MB class works by wrapping any given behaviour with delayed-retry functionality, exception handling and a scoping (Pad) facility.

6.4.1 MB

The ManagedBehaviour (MB) provides a manageability wrapper around any given behaviour. The class implements a basic retry-delay model and provides scoping functionality to the wrapped behaviour. The MB itself performs checks to determine the wrapped behaviour's state, capture exceptions, and manage the retry process. Most standard JADE behaviour functions are simply passed through to the wrapped behaviour. For an example, consider the MB.action() method.

A JADE behaviour's action() method is similar to the run() method in a Java thread. The JADE scheduler, having selected a behaviour to run, calls the behaviour's action() method. The action() method generally serves as a combination entry point and mini-scheduler that tracks the most recently completed phase of the behaviour execution. This tracking is necessary because JADE behaviours do not retain execution state between 'quanta' (in contrast to threads, which pick up precisely where they left off). The MB's action method performs some minor administration tasks and then passes control to the MB's nested behaviour, that actually performs the work. This process is illustrated in Figure 6.5.

6.4.2 MBMB

The other cornerstone of the behaviour extensions is the Managed Behaviour Managing Behaviour, or MBMB. The MBMB adds exception handling and certain operational niceties to the JADE FSMBehaviour. In particular, it provides:

- Prebuilt pause and cooldown behaviours
- Optional "Run-through" scheduling
- Exception-catching at the granularity of the JADE scheduler
- Pad functionality
- Simplified subbehaviour registration

MBMBs give the user greater simplicity and flexibility in developing complicated behaviours. In computations requiring intricate manipulation of many shared data structures, it is not uncommon to find behaviours nested four or

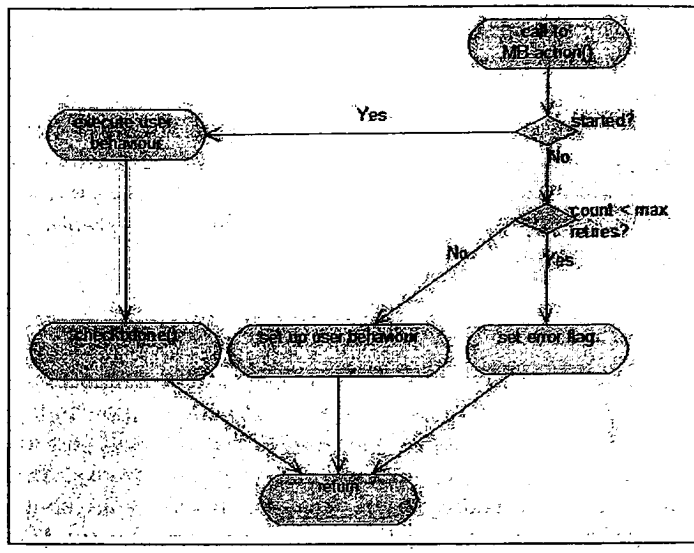


Figure 6.5: The Managed Behaviour `action` method logic.

five deep. Should one of these sub-behaviours fail, the user would probably want to know why. The MBMB, used in conjunction with MBs, retains information about the original exception that enables more traditional, 'Java-like' exception handling. Exceptions thrown in subbehaviour C percolate up the behavioral 'call stack', causing retries or failures in containing behaviours B and A in turn.

In addition to exception handling, MBMBs provide greater control over the JADE scheduler. JADE schedules behaviours in a round-robin, cooperative fashion. For smooth system operation, each behaviour must yield its 'quantum' by returning from the `action` method in a timely fashion. It was discovered that it is occasionally inappropriate for a behaviour to yield its quantum. For example, consider the case where an MBMB has as its subbehaviour a LockList-Behaviour and a user-defined behaviour that adds elements to the locked DAM. If the elements are ready to be added, yielding the quantum during this process only results in retaining the resource lock for an extended period of time. With the current write mechanisms, the delay translates directly into failed behaviour executions, especially when competition for the locked element is fierce.

The MBMB contains a variable called 'runThrough' for cases such as this. Setting the variable to true ensures that even subbehaviours written to yield their quanta run to completion. This is implemented in the MBMB scheduler, which re-calls the subbehaviour's action() method until it is complete. This has the effect of giving subbehaviours an endless stream of 'virtual quanta'. If used judiciously, run-through can increase critical resource availability in a Net.Sense system. If used carelessly (for example, to add 10,000 large elements to a DAM

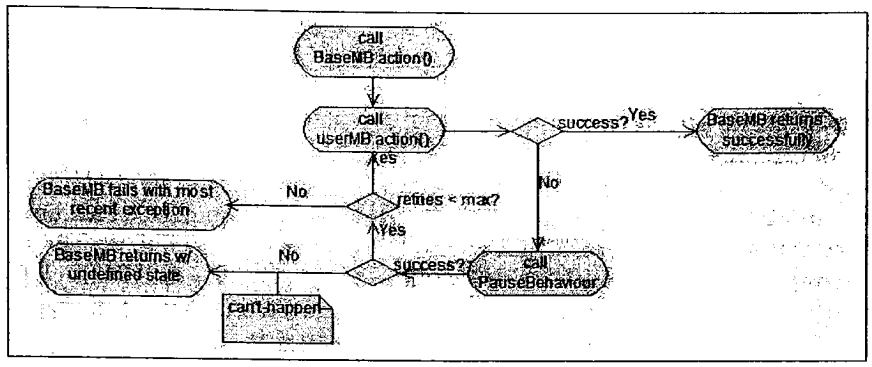


Figure 6.6: The BaseMB control flow

all at once), it can bring an agent to its knees, unable to respond to messages or generate its internal events. For this reason, run-through is controlled by a protected member variable that has no mutator. Child objects may modify their own runThrough state, but should not be able to modify that of others.

Finally, MBMBs provide convenience in constructing customized behaviours. The user need implement only three functions defined in the `jade.core.behaviours.Manageable` interface to enjoy automated behaviour registration, exception handling, pad services, and more. MBMBs serve as the foundation class for much of the NSBehavior heirarchy, including derivatives of GenericOpBehaviour.

6.4.3 BaseMB

The BaseMB class adds 'delay-and-retry' semantics to preexisting user-defined MBs. To do so, it first executes the user MB, and checks the result code. If the user MB succeeds, the BaseMB exits successfully. If not, the BaseMB passes control to a PauseBehaviour (see 6.2.2) that halts the behaviour for (at least) the delay time specified in the BaseMB object. When execution resumes, the BaseMB determines whether the user MB may be reattempted. If so, the process begins anew. If not, the BaseMB returns with a failure code and the exception recorded from the last execution of the user-defined MB. See Figure 6.6.

6.4.4 Pads

One of the more conceptually challenging aspects of the development effort was keeping track of a given object's position in what is colloquialized as the instantiation or containment hierarchy. Consider classes A and B. Let B inherit from A. Let B take a constructor argument of class A. Then the user may nest constructor calls as follows:

```
new B(new A(),new B(new B(), new B()), new B(newA()))
```

When using wrapper classes, it is difficult if not impossible to pass data between two or more general behaviours within the containment hierarchy. The Net.Sense Pad system implements a general solution to this communication problem. To do so, it captures the runtime containment hierarchy, and uses it to define scoping rules for the behaviours in question.

A Pad is a HashTable retrofitted to behave like a local scope. 'Variables' are stored in the hash table by mapping the variable name to an object containing the variable's value. The HashTable methods are overridden such that variables not found in the local scope are forwarded to the parent scope (if any) for resolution. Pads know their parent, and therefore variables declared in a given scope A are visible in all subscopes of A. The reverse is not true, however: a subscope's declarations are invisible to the parent, because Pads do not know their children. While this implementation has the advantage of providing 'local' variables, it has the consequence that variables to be modified (or filled) by inner behaviour A for later use by outer behaviour B must be declared in B's Pad, or a Pad in one of B's containing parents. See Figure 6.7.

The Pad system allows for the effective use of the Generic Op Behaviour templates. By declaring a 'variable' in the GenericOpBehaviour's pad, its sub-behaviours (acquire, lock, etc) can manipulate and share the variable. This permits greater modularization as behaviours become increasingly similar to distributed functions, with parameters and predicates.

6.5 Distributed Data Structures

At the heart of the InfoSpace lies the data structures that populate it. This section describes the structure, function, and life-cycle of the DDSs that live in the prototype InfoSpace. The discussion assumes basic knowledge of the concepts and usage of JavaSpaces technology, with which the prototype InfoSpace was developed.

The classes that make up the DDS hierarchy are depicted in Figure 6.8.

6.5.1 Root Interfaces

InfoSpace data classes are rooted in two interfaces: `Storable` and `DAE`. The Storable interface fills a gap in JavaSpace functionality by providing expiration dates for objects. Storable objects know how much longer they are to 'live' in the JavaSpace. This knowledge allows Net.Sense to create transient or ephemeral data – data with an 'expiration date'. This is particularly useful for raw sensor data: as the swarm collects data, Net.Sense accumulates a time-limited data history in the InfoSpace.

The other root interface is the distributed array element (`DAE`). Objects implementing this interface provide functionality to manipulate a data source designator, a matchable template (used te retrieve elements from the JavaSpace), a unique identifying tag, a data type, and a creation date.

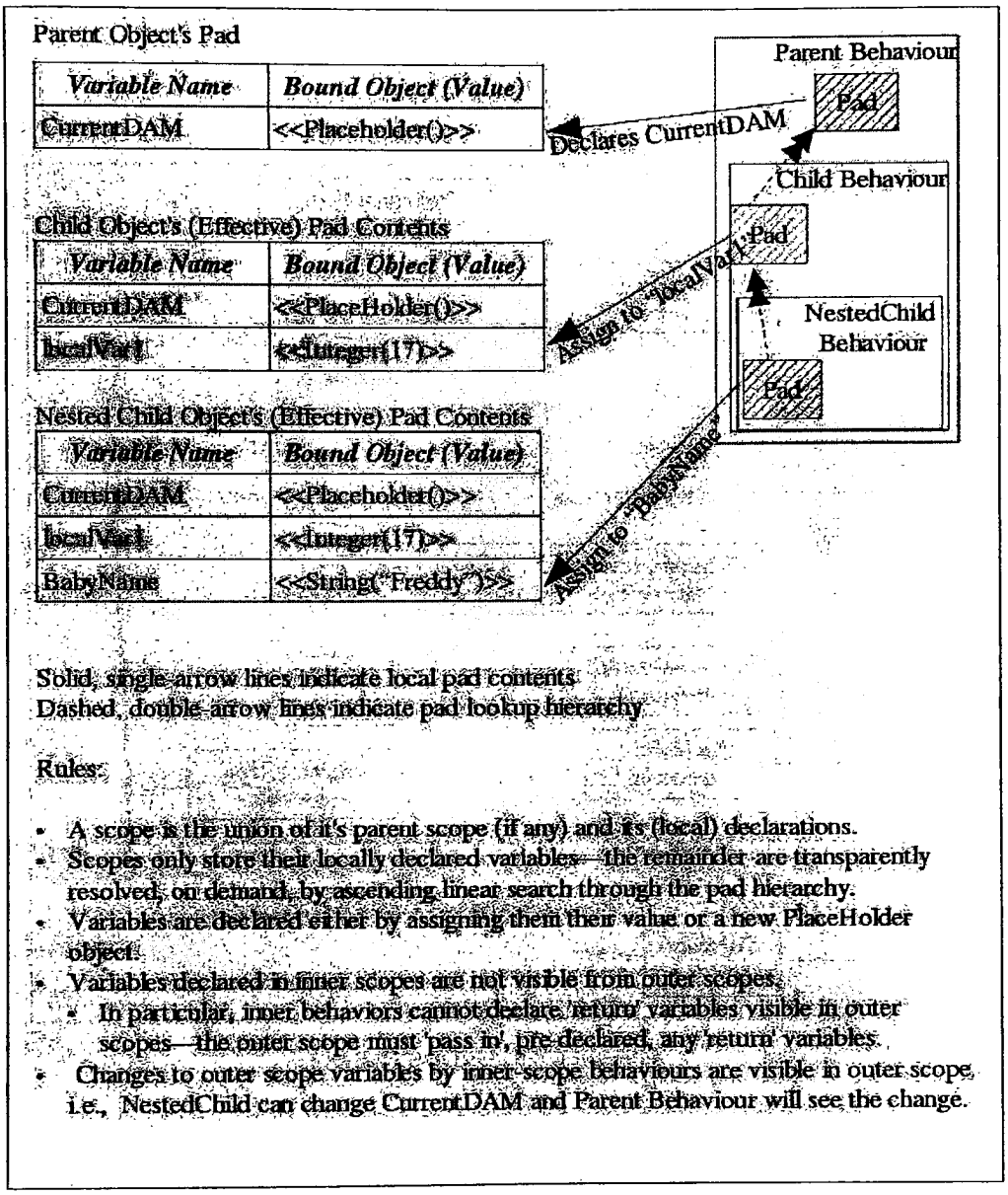


Figure 6.7: The Pad System

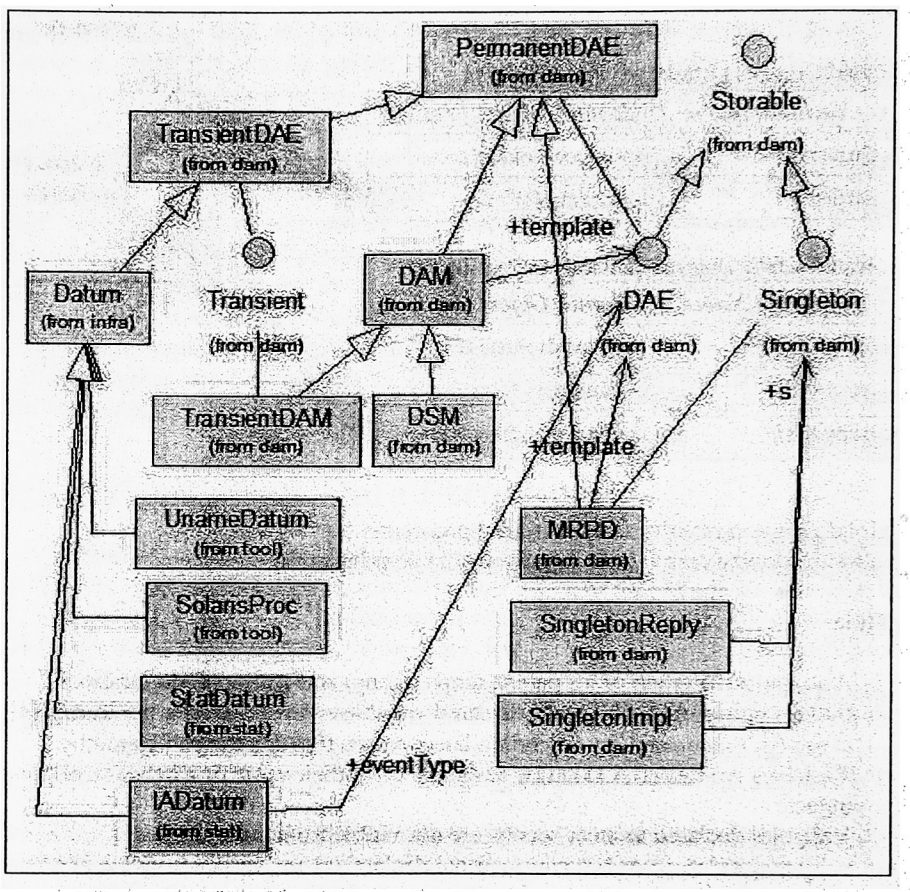


Figure 6.8: The DDS Hierarchy

The present implementation provides both transient and permanent varieties of DAE. The former are suitable for automatically collected, automatically expiring information such as raw sensor data. The latter are better suited for more permanent constructions that humans (user or programmer) will have direct and continued interest in. (For this reason, permanent DAEs are manually named at creation time, while transient DAE names are generated automatically.)

6.5.2 Implementing Classes

At the root of the implementation hierarchy lies the concrete `PermanentDAE` class. It provides basic equality semantics, template construction, ordering semantics (based on creation time), source information, unique tags, and lifetime information. Because this class models objects with infinite lifetimes, the `getRemainingLeaseTime()` method returns JavaSpace's `FOREVER` lease time. (TransientDAEs override this method to provide 'genuine' durations.)

From PermanentDAE are derived three children: `TransientDAE`, `DAM`, and `MRPD`. The first two are concerned with data storage within the InfoSpace, and the last is metadata that assists with the processing of that data.

The `TransientDAE` class provides functionality mandated by the `Transient` interface that provides 'proper' expiration dates. Expiration dates are computed based on the (then-current) date and the desired Datum lifetime. The `DAM` class provides list-like functionality within the InfoSpace. Owing to the complexity of the DAM class' functionality, discussion of the DAM and its children are deferred to Section 6.5.3.

Last of the three children is the Most Recently Processed Datum (`MRPD`) class. It provides placeholders for session-based processing of live data sets. The MRPD associates a date and DAE template with a name. When a processing behaviour is scheduled, it simply retrieves the MRPD, examines the time contained therein, and continues work where it left off. At the conclusion of a processing session, the behaviour updates and returns the MRPD.

MRPDs implement the Singleton interface. As such, the InfoSpace should contain at most one MRPD with a given name at a time. See Section 6.5.4 for more details.

The `Datum` class is found at the next level of the class hierarchy. It is primarily responsible for storing information about where its encapsulated datum was collected, and interfacing that datum with the Java Expert System Shell (JESS). Test development with JESS provided rudimentary population control in a Net.Sense system. While this validated an aspect of the Net.Sense vision, subsequent changes to the InfoSpace API precluded that code from operating in the current implementation.

6.5.3 DAM & DSM

Overview

To their users, instances of the DAM family function like distributed lists, or distributed sets in the case of DSMs. Internally, however, their structure differs significantly from traditional List or Set implementations. DAMs are built around HashMaps that relate positional markers to object meta-data. Net.Sense DAMs do not actually contain the user's serialized data objects. Rather, they contain and relate references to these data objects in the JavaSpace. This indirection gives DAMs significant advantages over traditional data structures stored in the JavaSpace.

- DAMs are 'JavaSpace-transparent'. Because DAMs do not encapsulate the elements they 'contain', the elements and their fields can be directly accessed (and, for example, matched on).
- DAMs are lightweight. For example, accessing the `all-processes` DAM in an operating testbed system requires space proportional to the number of hosts in the network, rather than to the number of processes currently stored in the InfoSpace. Were the lists of processes encapsulated in their containing lists, accessing the `all-processes` DAM would require deserializing every process list in the JavaSpace!
- DAMs enable parallel access. Multiple agents can simultaneously obtain write access to distinct elements in a DAM because DAMs can be locked (removed from the JavaSpace) individually.

To understand these advantages in context, one must first recall a key point in JavaSpace's design, namely that objects to be put into a JavaSpace are serialized and stored in the space as inert bit strings.

Transparent

A consequence of the 'flat' nature of the JavaSpace storage mechanism is that method calls *cannot* be made upon 'objects' within the space, but only upon objects locally reconstituted from the JavaSpace's serialized representation.

This poses significant challenges for the developer. In particular, equality in the JavaSpace is determined not by the `equals` method but by comparing the serialized representations of each field in the stored object. Were a DAM simply a wrapper around a Java List, it would be infeasible to examine or search for elements within DAMs. Doing so would entail deserializing the entire DAM and all its nested Objects. For large meta-structures, this would be quite impractical. DAMs provide enhanced efficiency by requiring deserialization of only the meta-data relating the list members, rather than the whole (potentially deeply nested) list.

Lightweight

When DAMs are modified, their implementation requires only meta-data to be read and written, improving the efficiency of adding and removing elements from DAMs.

This decision has a negative consequence, however. Iterating through a DAM requires a series of JavaSpace accesses, rather than a single, large read. This is because JavaSpaces have no query mechanism that returns multiple results. The performance consequences of this have not yet been assessed.

Parallel

DAMs are structured to permit parallized access to their members. Distinct accessors interested in distinct elements of a DAM can operate on their chosen elements simultaneously and in parallel. In database terms, the current Infospace implementation provides functionality akin to record-level locking for reads. Unfortunately, element modifications are currently performed by temporarily removing the DAM undergoing modification from the JavaSpace. This has the effect of reverting to a writer-favoring version of table-level locking when adding to or removing from a DAM.

This effect is simply explained: reads and iterations can be performed without removing the DAM from the JavaSpace. DAM modifications, however, are implemented by removing the element from the space, changing it locally, and returning it to the JavaSpace. While the DAM is out of the JavaSpace, other readers cannot access it.

Functionality

As mentioned earlier, DAMs provide functionality by mapping positional keys to matchable template values, called tabs. Tabs are matchable template values, and are named after their children's book counterparts: when 'pulled on' via a `read` or `take`, the desired object 'pops up'.

Element addition

When a new element is added, the DAM finds an unused positional key and associates it with the new object's tab. Once the mapping is made, the element is automatically written into the JavaSpace unless it is a DAE subtype that should already exist in the JavaSpace. Examples of such subtypes include DAMs and Singletons. At present, the logic performing this determination is hardcoded into the DAM class.

Iteration

To iterate over a DAM, the user must first create an iterator. DAM iterators are, internally, unidirectionally traversable snapshots of the DAM's tab array. Iteration is accomplished by stepping through this array of tabs and reading the corresponding elements from the JavaSpace.

Element Removal

Removing an element from a DAM is accomplished by simply removing the position-tab mapping. As DAEs can be permanent and/or members of multiple DAMs, DAEs removed from a DAM are not removed from the JavaSpace. Leaving the element in the JavaSpace poses no significant problems because either the element is permanent (in which case it should not be removed), or transient (in which case it will expire automatically and be removed by DAMFactory garbage collection).

Life Cycle

DAMs are created by a DAMFactory, of which each Net.Sense prototype system must have exactly one. The DAMFactory is presently implemented as a multi-threaded standalone application. Its two responsibilities are to service DAMRequests and to periodically garbage-collect expired DAMs from the JavaSpace.

The DAM creation process is relatively simple. Through a DAMConnector, a requestor writes into the JavaSpace a new DAMRequest. The DAMFactory is notified and spawns a thread to extract all pending requests from the JavaSpace. For each request, the factory determines whether the requested DAM is permanent or transient. The desired DAM is then created, added to the JavaSpace, and a reply is sent to the requestor. In the event that the requested DAM was permanent, the factory verifies prior to creation that a DAM with the same name does not already exist.

Periodically, the DAMFactory iterates over its collection of permanent DAMs. Each DAM is temporarily removed from the space, and its elements are iterated over. Any element inaccessible for more than a certain interval is removed. This rather crude expiration policy suffices for experimentation but creates the potential for data loss. If DAM A is contained in DAM B, but is locked for longer than the removal interval while DAM A is compacting, DAM B can be incorrectly removed from DAM A. This is a known issue with the prototype implementation.

6.5.4 Singletons

Singletons are shared data structures whose defining characteristic is that they must be unique throughout a Net.Sense system. Unlike DAMs, which are likely to serve without extensive subclassing, the Net.Sense Singleton architecture was designed with the idea that users would frequently write their own Singletons. To that end, the Singleton implementation attempts to maximize the user's development options.

6.5.5 Functionality

The present Singleton implementation provides basic functionality that users will expand upon to provide functionality needed for specific applications. The prototype framework provides a useful example: the MRPD, or Most Recently Processed Datum. Objects of this type keep track of the newest processed list element. This enables users to write code that can maintain its place while processing a 'live' dataset.

Creation

Like DAMs, elements of the Singleton framework are created by a single, threaded Factory through a request process facilitated by a connector class. Like DAMs, Singletons are also named. Unlike DAMs, however, Singletons can be also categorized into groups.

Rather than requiring subclasses to provide a myriad of constructors, the Singleton implementation requires a default constructor and a configuration method that takes an Object array of parameters with which to set up the Singleton object. The present implementation requires that the first two elements of that array be String objects containing the Singleton's name and group, respectively. Should implementors elect to wholly override the parameters as desired. However, it is desired that they either keep to the established convention, or standardize one.

As with DAM creation, clients use connectors to create and send requests that are serviced by the factory. The factory in turn produces replies that the client uses to access the newly-created Singleton object. It is the factory's responsibility to ensure that only one Singleton with a given name is created.

The current implementation has a known denial of service vulnerablility in that broken or malicious agents could remove Singletons (or permanent DAMs) from the InfoSpace, and never return them. These 'stolen' structures would be lost and never recreated, crippling the Net.Sense system.

6.6 Agents

The Net.Sense vision calls for an agent chassis that is immutable as of compile time. This chassis supports the loading of toolkits and behaviours that contain code providing specific abilities to agents. In the prototype implementation, the chassis is actualized in the `ns.infra.BaseAgent` class. It provides convenience behaviors over and above those provided by the JADE agent class. In particular, the BaseAgent class provides centralized message handling, recurring internal message generation, limited runtime behavior and tool loading, exception handling/logging, automated, 'safe' movement, and integrated access to the JavaSpace. The present implementation does not support dynamic behaviour (un)loading.

6.6.1 NSB and Tool Management

From the agent perspective, Tools and NSBs are similar, as the Net.Sense vision states that both should be dynamically (un)loadable. While the prototype framework does not yet fully realize the vision in these areas, it does provide basic behavior and tool management in the form of the NSBMaster and Toolbelt classes. These classes are responsible for loading, registering, and controlling the activation of NSBs and tools, respectively.

Net.Sense behaviors are maintained by the Net.Sense Behavior Master, or NSBMaster. This class is responsible for all agent-visible aspects of NSB administration. The NSBMaster provides agent authors the ability to load and register NSBehaviors with a single line of code. Behind the scenes, the Net.Sense framework automatically registers the new NSB's recurrent messages, registers the NSB's services with the agent framework, and makes connections enabling the new NSB to receive messages and be checked for activity.

Net.Sense Tools are collected and managed by the `Toolbelt` class. The Toolbelt's internal responsibilities are much simpler than those of the NSBMaster, as Tools do not directly receive messages or provide services. Rather, the ToolBelt serves as an interface between the agent's NSBs and the actual tools. It provides two functions: determining which, if any, tools are active, and fetching loaded tools for NSBs.

6.6.2 Messaging

via JADE

The JADE messaging system provides both inter- and intra-agent communication capabilities. Client code wishing to communicate creates an ACLMessage object, fills its fields appropriately, and calls the Agent.send() method. Agent behaviours wishing to receive messages must each write code that polls for messages via the Agent.receive() method, and only then may act. This complicates behaviour scheduling as agent behaviours must be active in order to receive messages. At the time of development it was felt that the simplest solution would be to implement a centralized messaging center. This communication hub is realized in the Comm Center, which provides a common access point for sending and receiving messages. The Comm Center reduces behaviour complexity, code size and the challenges of debugging agent communications.

The Comm Center is realized in the `infra.CommCtr` class. Its two methods, Send() and Receive(), wrap calls to the underlying agent's JADE-given methods. Universal message receipt is accomplished by a `ListenBehavior` object that repeatedly calls the Comm Center's Receive() method and thereby the JADE agent's blockingReceive() method. (Note that ListenBehavior is a behaviour in the JADE style, and so its name should be spelled with a 'u'.)

The Comm Center / ListenBehaviour design efficiently multiplexes message handling with user code. Behaviours simply register themselves with the Comm Center and automatically receive copies of incoming messages whenever they arrive. At present, the Comm Center is a 'broadcast' medium in that all incoming messages are relayed to all listeners.

via JavaSpace

In the JADE framework, all meaningful agent interaction takes place through messaging. In Net.Sense, JADE messaging serves more as an event-notification system than a primary means of interagent communication system. This is so because of functionality or convenience issues arising from the volume, content, processing, and/or storage requirements of the communications. When the Net.Sense system needs to move large quantities of data, data residing in custom objects, or data for which processing can be parallelized, Net.Sense transmits that data via the JavaSpace.

As an example, consider the DAM creation protocol. An agent publishes a DAMRequest into the JavaSpace. This request contains the information needed

for the factory to construct a DAM, plus a unique identifying token. When the DAMFactory has created a new DAM, the factory publishes a reply to the request into the JavaSpace. The reply contains the DAM and the request's identifying token. The client retrieves the DAMReply containing the token it sent and extracts the DAM inside, thus completing the exchange.

The key advantages of the JavaSpace system are its asynchronicity and scalability. Factories, being responsible only for reading and servicing requests, have no direct interaction with any client. Thus the protocol implementation requires only a thin layer of management code. The client, meanwhile, is free to implement whatever polling strategy is most appropriate for its application. Further, the loosely-coupled nature of the JavaSpace communication medium allows for seamless, transparent scaling of DAM creation from one factory to a cluster of many.

Though not fully exploited in the present implementation, the potential exists for the JavaSpace-based performance of a variety of Net.Sense functionality. In particular statistics computation, interarrival times, and even DAM and Singleton creation could potentially harness JavaSpace task channels to transparently task automatically-scalable agent communities.

Recurrent event handling

Users often wish to write code to take action when a particular event occurs. Like JADE, Net.Sense uses messaging to provide event notification services.

One of the most fundamental aspects of the NetSense interagent control system is the notion of a recurrent event. A recurrent event is simply an event that should automatically recur at predefined intervals. When the agent's capabilities are viewed as event handlers, recurrent events can be seen as automated event schedulers. Event handlers (in Net.Sense, NSBehaviors) are responsible for determining when and how to respond to event notifications, preserving the agent autonomy criteria.

The Net.Sense prototype uses recurrent messages in a variety of ways. Swarm movement, IAT computation, and statistics computation are all triggered by recurrent messages. The IMGenerator class provides intermessage delay and permits users to easily configure delay-based messages by scheduling RecurrentMessage objects. RecurrentMessages are JADE ACLMessages extended to manage the interval between messages.

6.6.3 Movement

In the testbed system, only swarm agents move. As a result, framework movement support is somewhat biased towards addressing swarm movement issues. In particular, it is assumed that agents are singly-threaded as in the JADE model, and that agent movement is random 'wandering'.

The swarm's movement needs are to periodically and randomly select and then move to a host in the managed network. This functionality is implemented

in the `ns.infra.WanderBehavior` class. From a high level perspective, it registers recurrent messages that instruct the agent to move. On receiving such a message, the behavior will check whether or not it is safe for the agent to move. Because no method is yet implemented to determine which operations must not be interrupted, the framework errs on the side of caution and refuses to move when any tool or behavior is active.

If no active tasks are found, the agent selects a host at random, and calls JADE's movement code. If active tasks are found, the agent sets flags that prevent the Toolbelt from activating tools. This is a logic error, as it is feasible that an active NSB may need to activate several tools to complete its task. Should the agent suspend tool activations before all of an active NSB's tools have activated, the NSB could potentially enter livelock.

6.7 Review

This chapter has provided an overview of the core classes comprising the prototype implementation of the Net.Sense framework. In addition, it provides examples and illustrations of many of the key processes and component interactions associated with operation of any given Net.Sense system.

Chapter 7

Testbed System

This chapter provides an overview of the testbed Net.Sense system developed with the prototype framework. It reviews the key classes, operational nuances, and data flows within the testbed system, and serves as background material for the experimentation in Chapter 8.

7.1 Overview

7.1.1 Scope

Time limits nontrivially constrained the scope of the testbed system. Unforeseen framework design and development issues required resolution before useful testbed development could proceed. As a result, the testbed system is neither comprehensive, nor infallible, nor suitable for use outside the experimental environment.

The testbed system's shortfall in operational value is balanced by its value in exposing design faults, scalability problems, organizational and logical flaws, and other issues with the prototype framework. Even disregarding its usefulness in supporting or refuting our hypotheses, and despite its operational failings, the testbed system's development was worthwhile precisely because it pointed out those failings' causes.

At present, the system collects from the managed network data about running processes. It then computes interarrival times and useful metastatics about these data. From a developmental perspective, the testbed system demonstrates proof of concept, explores framework usability, and enables performance data collection from which conclusions regarding practical usability and potential improvements can be drawn.

7.1.2 Design Overview

The testbed system contains two of the three agent communities: sensors and reasoners. The sensor community consists of a group of `stat.Bug` agents whose

mission is to wander the managed network and collect process listing ('ps') information from the hosts therein. The reasoner community consists of two types of agents: IATAgents and DStatAgents. These two agent types examine data provided by the bugs and compute metadata useful to the system administrator in monitoring the QoS provided to the managed network.

This section provides an itemized overview of the key operational classes in the testbed system. For detailed discussion of the operation of and interaction between the classes below, see Sections 7.2.2 and 7.2.3.

The key classes in the test system are as follows:

CIATB & IAAcquire

The Compute Inter-Arrival Time Behaviour (CIATB) class is a specialized version of the CASSBehaviour class discussed in Section 6.2.1. The CIATB's chief distinction is that it overrides the normal CASSBehaviour acquire phase with an InterArrival Acquire (`IAAcquire` class, IAA for short) behaviour object. The IAA behaviour is responsible for getting the requisite most recently processed datum (MRPD), creating it if necessary. See Figure 7.5 and Section 7.2.2 for details.

DSCBehaviour

The Descriptive Statistics Computer Behavior (DSCB) implemented in the `DSCBehaviour` class is a subclass of the Simple Tool Invoker Behaviour or STIB, discussed in Section 6.3. `DSCBehaviour` (a misnomer: as a subtype of NSBehavior it should not be spelled with a 'u') is a rather intricate NSB as it provides three different modes of operation, according to the source(s) and destination(s) of the data being processed. The intricacies of this class' operation are discussed in detail in Section 7.2.3.

IATBehavior

The IAT Behaviour class (`IATBehavior`) controls the IATAgent's instance of an IATComputer tool. As this tool has one primary operational mode, the IATBehavior is correspondingly simpler: its responsibilities are effectively to extract the names of all items in a given DAM, format them suitably for configuring an IATComputer tool, and then activate that tool.

IATComputer, CIB, DIB, & FWB

The IAT Computer tool provides an agent the capability to compute interarrival times between data elements in the InfoSpace. The computed times are stored back into the InfoSpace where they can be reasoned over. Helper classes such as Determine IATs Behaviours (DIBs) and Find Work Behaviours (FWBs) are employed by the IATComputer to manage the complexity of IAT computations (which involve one source and three destination lists as well as an MRPD).

DIBs are the high-level task representation within the IATComputer. Each tasked source list is used to create a DIB that captures the process flow required to turn a source list into an updated list of IATs. DIBs orchestrate the subsequent preparation for and execution of FWBs and Compute IAT Behaviours (CIBs). FWB's are responsible for determining which, if any, of a given list's elements are ready for IAT computation. The `CIB` class performs the actual work of computing IATs while taking into account issues like network lag.

StatDatum & IATDatum

The IATDatum is a `Datum` subtype that binds an event type to a measured interarrival time, in milliseconds. IATDatums implement the SimpleNumValue interface, which means that they have a default value (the IAT) that can be automatically extracted by tools such as the DStatComputer.

The StatDatum is a `Datum` subtype that contains descriptive statistics about a given data set. In the present implementation, these statistics are the set's maximum, minimum, mean, variance, standard deviation, and size (element count). It also provides SimpleNumValue functionality that defaults to providing the 'mean' field. This decision limits the usefulness of the metadata processing that can be performed. Consider the administrator interested in finding the mean *maximum* IAT for a network. This is a known issue that will be resolved when an acceptably flexible and elegant solution is found.

StatKeeperAgent

`StatKeeperAgents` are responsible for providing most of the metadata in the testbed system. Each is equipped with a DStatComputer tool and a DSCBehaviour configured with a StatParamSet that drives the sequence of metadata computations described in Section 7.2.3. While the present testbed system only utilizes one DStatAgent, future releases will provide statistical computation services as a service of the Reasoner community, as called for in the Net.Sense vision.

Bug

The testbed system's Sensor community is made up of Bug agents. Each mounts a ProcessLister Tool, a Simple Tool Invoker Behaviour (STIB) and a WanderBehaviour. The WanderBehaviour requests the agent to move to a random host in the managed network at fixed intervals. Upon arrival at a new host, the Bug sends its STIB a message to fire the ProcessLister Tool. The ProcessLister executes a 'ps' command on the host and parses the output into SolarisProcess objects. The SolarisProcess objects are added into the JavaSpace, where they can be found by the IATAgent.

DStatAgent

The Descriptive Statistics (DStat) Agent serves as the focal point for statistical computation in the testbed system. The agent mounts one Tool, a DStat-Computer, and one Behaviour to control it: a Descriptive Statistic Computer Behaviour (DSCB). This controller behaviour schedules three recurring events, one for each phase of the data reduction process. The first produces statistics for each host's IATs. The second summarizes those statistics into network-wide snapshots. The third summarizes those snapshots over time.

IATAgent

The IATAgent computes interarrival times for arbitrary classes of events. The agent mounts an IATComputer Tool and the IATBehavior that controls it. The IATAgent schedules one recurrent event, that fires its IATBehavior with instructions to process the InfoSpace's 'all-processes' DSM, creating the metadata that form the base of the QoS analysis process.

7.2 Operational Processes

The author found it exceedingly easy to 'lose his place' when thinking about the processes for computing metadata. In particular, maintaining a firm grasp of precisely what is being computed and what it means in context. To clarify the rather complex process, we step through the testbed system's operational processes. To ensure reader comprehension, metadata computation and data flows are described in detail.

7.2.1 Process Collection

As in the Net.Sense vision, the swarm (a community of Bugs) is responsible for collecting information about what processes are running on the managed network. The Bugs wander independently around the managed network, examining the processes running on the local host at each visit.

When a Bug arrives at a new host, it activates a Behaviour that fires the Bug's ProcessLister Tool. This tool starts a 'ps' process on the local host, and captures the output stream into a buffer. The buffer lines are parsed into SolarisProcess objects that are collected into a list.

At this point, the Bug must put the collected data into the JavaSpace. This requires three actions. First, the agent must create a transient DAM to hold the current batch of processes. Second, the agent must add the new DAM to the host DSM, which contains this host's transient process DAMs. Third, the agent ensures that the host DSM is a member of the global collected-process DSM. (Figure 7.1) Figure 7.2 shows a graphical representation of the completed data structure in the JavaSpace.

At present, these three steps are taken by the ProcessLister Tool. While effective, it is neither particularly good software engineering nor part of the

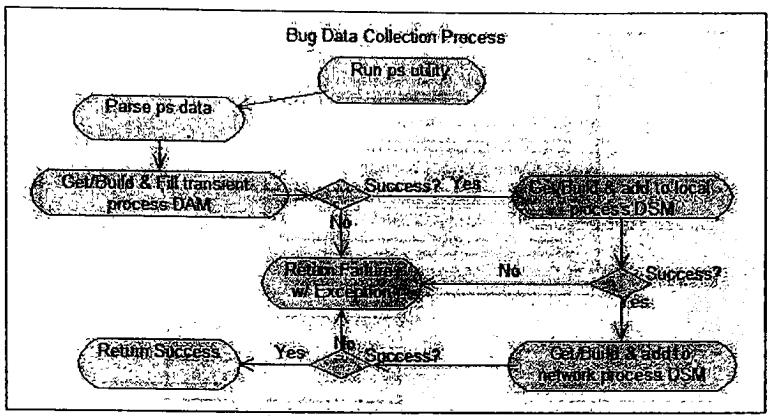


Figure 7.1: Data Collection Process

Net.Sense vision. In the future, it is hoped that further effort will be devoted to separating and encapsulating the data storage and access routines (See Section 9.2.1). This should facilitate writing new sensors and modifying the D[A,S]M storage hierarchy.

7.2.2 InterArrival Time Computation

The testbed system's IATAgent is responsible for computing interarrival times for incoming data. Interarrival times are the intervals between *visits to a given host*. A visit is defined to be any time any Bug arrives at a host, recovers process data from that host, and adds it to the JavaSpace. Note that agents do not visit 'hosts' but rather they visit 'containers' that run on those hosts. For Net.Sense purposes, it is assumed that each host machine in the managed network have exactly one container.

Theory

Definition 7.2.1. *IAT_χ*

If a container/host χ is visited at time t_1 by agent A, and at some time t_2 by Agent B (where B is not necessarily distinct from A), then an interarrival time for host χ, abbreviated iat_χ, is

$$iat_\chi = t_2 - t_1 = \Delta t$$

Consider a network with two hosts χ_1, χ_2 and one agent A. Suppose that, as time passes, A repeatedly alternates between hosts 1 and 2. Then, if A visits χ_1 at times $t_1, t_2, t_3, \ldots$ then the set of interarrival times is

$$IAT_\chi = \{t_2 - t_1, t_3 - t_2, \ldots\}$$

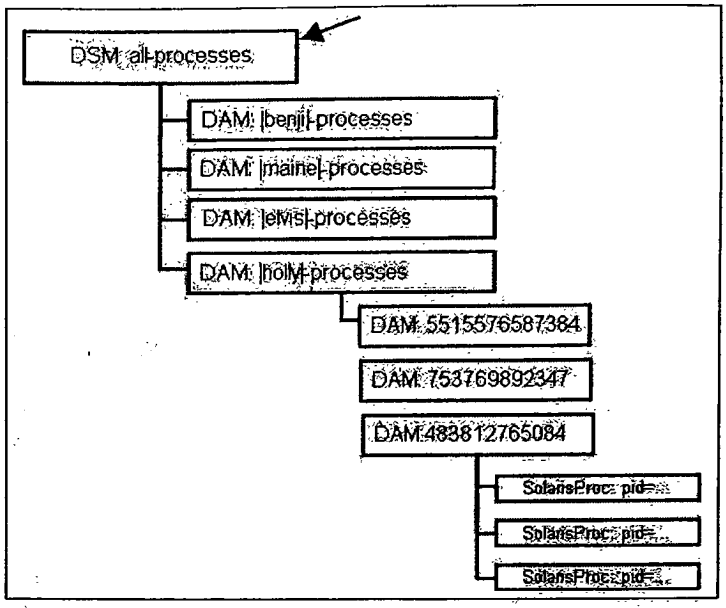


Figure 7.2: Process Storage Hierarchy

This set contains raw information about how frequently host χ is visited. If we consider a network with n machines $\{\chi_1, \chi_2 \cdots \chi_n\}$,then we are naturally interested in computing interarrival times for each host. All sets of IATs are collected into

$$AllIAT = \{IAT_\chi^1, IAT_\chi^2, IAT_\chi^3 ...\}$$

for convenience in later processing.

Implementation

The IATAgent lies dormant until its IMGenerator sends a Poll message. This message is received by the agent's IATBehaviour, and it tells the agent's IATBehaviour to process the 'all-processes' DSM. The IATBehaviour creates a Basic List Operation Behaviour (BLOB) with a new `IATBehavior.PMLB` as its user-defined behaviour. The PMLB, or Process Meta-List Behaviour, is a JADE-type behaviour that retrieves the 'all-processes' DSM from the JavaSpace and iterates through it, collecting the names of the lists within. These, inner lists contain the actual events for which IATs are to be computed. In the testbed's case, the events are the creation of process data snapshots for hosts in the managed network.

The list of names is passed to the IATComputer, which sets in motion a complicated process that sets up the necessary infrastructure to compute IATs.

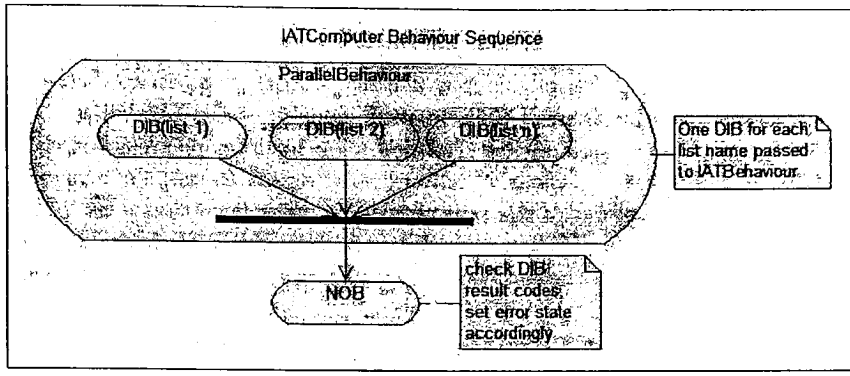


Figure 7.3: IATComputer Control Flow

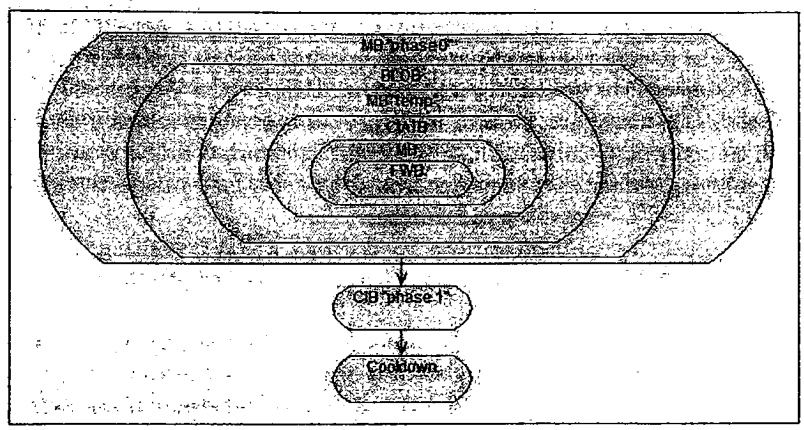


Figure 7.4: DIB Structure

The IATComputer constructs and schedules the top-level behaviour hierarchy as shown in Figure 7.3.

The behaviour hierarchy is a SequentialBehaviour that first executes in parallel a DIB for each tasked data series. Upon their completion, the SequentialBehaviour calls a Notification Behaviour (NOB) responsible for collecting results (IATs), setting status conditions, and cleaning up. The actual work is performed by the DIBs.

The DIB itself is a somewhat complicated construction by nature of the subbehaviour wrapping that takes place. The reader may find it helpful to refer back to Figure 7.4 while reading through the following discussion. For simplicity, we will begin at the kernel, the Find Work Behaviour (FWB) and work outwards toward higher-level behaviours.

The FWB at the core of the DIB performs one simple task: enumerate any

new data that can be processed safely. The FWB accomplishes this by reading elements out of the DAM, sorting them, and selecting 'appropriately dated' elements. These elements are put into a list for further processing, and an MRPD is updated with the time of the newest element in the result list.

Unfortunately, the criteria for determining what is 'appropriately dated' are slightly more complicated. Clearly, only new *intervals* should be processed. This requires that the newer of the pair of last elements be retained as 'unprocessed' so that it is available when new elements arrive. While this adequately defines the trailing edge of the data, choosing the leading edge requires special consideration.

JavaSpaces do not guarantee proper sequencing of additions to and removals from the JavaSpace. Therefore, objects created (and added to the JavaSpace) at time t can appear in the JavaSpace before items created and added at times prior to t. Should such an occurrence be commonplace, the IATs will be materially skewed as events will appear to occur farther apart than in reality.

To reduce skew, the FWB incorporates a fixed 'settling period' to permit latecoming data to join the JavaSpace before being used as a basis for an IAT. This delay is presently set at three minutes, 10% of the default 30 minute Datum lifetime. This figure is quite arbitrary, and experimentation should be performed to determine the relationship between network conditions and JavaSpace sequence irregularities. Such experimentation is far outside the scope of this thesis, and will not be discussed further.

To summarize, the selection criteria applied by the FWB are that data be no older than the MRPD date and no younger than three minutes from the FWB creation date.

FWB execution requires a DAM containing the data set to process and an MRPD to 'hold its place' with as described above. It is the task of the many wrapper behaviours surroundng the FWB to acquire (and create if needed) these items for the FWB. Moving outward, the first layers are an MB and a CIATB, respectively. For information on the MB's role, see 6.4.1. The Create IAT Base Behaviour (CIATB), is responsible for acquiring (and creating, if necessary) the MRPD needed by the FWB. The CIATB is a subclass of CASSBehaviour that overrides the acquire state with a custom IAAcquire object. Its control flow is depicted in Figure 7.5.

The custom acquire phase (IAAcquire object) used by the CIATB first makes repeated attempts to locate an existing Singleton with the given name. If no Singleton by that name can be found, the IAAcquire concludes that the requested Singleton does not exist, and attempts to create a Singleton with parameters as generated by the `getSR` routine. These parameters set the MRPD date to be the earliest possible date to ensure that no data are skipped as being before the cutoff.

Returning to the diagram, the reader should note that an MB and a BLOB enclose the CIATB. The MB provides the customary management functionality as described in Section 6.4.1. The BLOB is responsible for acquiring the DAM over which the FWB operates, and incidentally serves as an escape clause for the nested CIATB. The BLOB fails if it is unable to acquire the requested list.

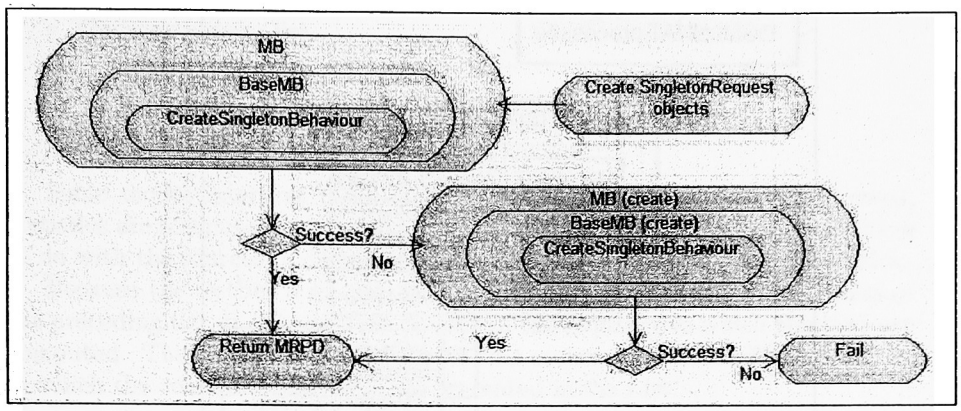


Figure 7.5: IAAcquire Control Flow

Because this failure is detectable, the overhead associated with the CIATB may be avoided if the requisite DAM is unavailable. This discussion demonstrates how wrapper classes may be applied to simplify behavour construction. From the outside, working in, the flow is:

- Acquire the needed DAM, or fail
- Acquire the needed Singleton, (itself a complex process) or fail
- Use the DAM and Singleton together to perform the needed processing, which remains in a self-contained module that can be easily subclassed and extended.
- (Automatically) return the modified DAM and Singleton to the JavaSpace.

Once the DIB has completed, control passes to a Compute Intervals Behaviour (CIB). This behaviour is also relatively simple. It has three phases: a compute phase followed by two phases responsible for building the hierarchy of DAMs that makes the results available to the statistics processors to be described later.

The CIB computation phase is rather simple, The behaviour simply loops through the List provided by the FWB, computing the differences between successive elements' creation times. IADatums (effectively wrappers around a Long object) are created from the resulting intervals. The CIB performs some legerdename with the hostName element of the IADatum. This changes the IADatum's hostName from the host where the IATAgent resides, to the host from whose data the IATDatum was derived.

Once the IADatums are computed, the CIB begins adding them to the JavaSpace [1]. The CIB first acquires or creates a DAM of IATs for the given host,

[1] As noted elsewhere, (See Section 9.2.1) this process should be factored out and standardized.

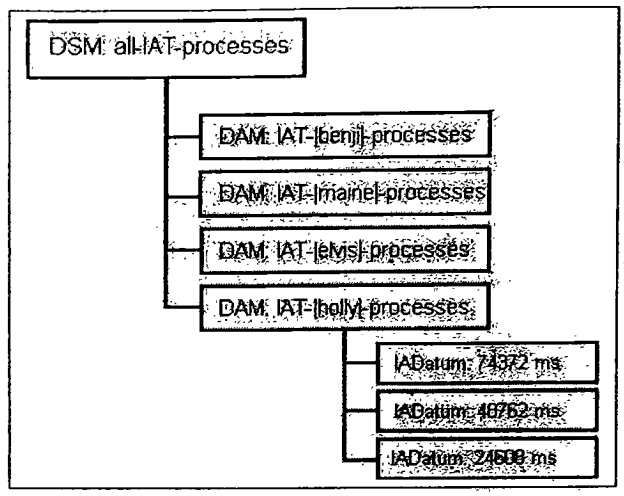


Figure 7.6: Interarrival Time Storage Hierarchy

and adds to it the newly created IADatums. This DAM is named by prepending "IAT-" to the name of the list from which the CIB's data is drawn. For example, computing IATs for a host named 'iowa' found in a meta-list named 'all-processes' are placed in a DAM named `'IAT-|iowa|-processes'`.

Finally, the agent acquires or creates a DSM to contain all hosts' IAT DAMs. This DSM is named by taking the internal DAM name, stripping off the hostname and prepending 'all-'. In the previous example, the DSM containing all managed hosts' IAT DAMs would be named 'all-IAT-processes'. This hierarchy is illustrated in Figure 7.6.

7.2.3 Descriptive Statistics Computation

IADatums, while useful, are too abundant for ongoing, detailed consideration by a system administrator responsible for tens or hundreds of hosts. To make this information practical, it must be rendered down into a handful of statistics that summarize the QoS trends in the managed network. To accomplish this, the testbed system summarizes and collates information into a more managably sized data set that describes the QoS provided by the testbed system to the entire managed network over time. This process begins with the set $All - IAT$ constructed by the IATAgent.

Before beginning, a general observation should be made about the DStatAgent and its use. The DSCComputer initiates work upon receipt of a command message list. Each command message is a LinkedList of four Strings. A command message is required for each distinct data set to be summarized. The strings are, in order:

- The operation mode (separate, aggregate, or individual)

- The source meta-list
- The destination prefix
- The meta-destination prefix

Some modes interpret these strings differently than others. In all cases, statistics are computed in the same fashion; the differences between modes have to do with the sources of data and the destinations of the computed statistics.

Because the testbed's second and third tier statistics rely on the results of the preceding tier, it is important to control the sequence in which the tiers are computed. The testbed system controls this sequence by controlling the order in which the command messages are presented in the List.

Phase 1: Current host statistics

The first phase of the reduction process determines, on average, how long a given host goes between visits. It computes basic descriptive statistics for each host χ's window of IATs IAT_χ. Host results are added to a brief DAM history for that host (notated $\overline{IAT_\chi}$). These host DAMs are kept in one master DAM much as was done when collecting processes.

This is accomplished by sending the DSCBehaviour a first command message: a LinkedList containing the 'SEPARATE' command defined in the DSCBehaviour, followed by "all-IAT-processes", "DS-", and "all-DS". This indicates that the DSCBehaviour is to use 'all-IAT-processes' as a task list. For each (host) DAM in that task list, the DSCBehaviour extracts the DAM's contents and computes from them a StatDatum for the given host. This StatDatum is added to a DAM named "DS-*sourceDAM*", which is placed into a DAM named "all-DS-*sourceDAM*". See Figure 7.7.

Phase 2: Historical host-based statistics

Phase 1 provided statistics for each host at a given point in time. The next step on the path to usability is to produce statistics for each host's snapshots. To do this, statistics must be computed for each element of $\overline{IAT_\chi}$, and then collected into a list. The command message to do this contains the 'aggregate' String defined in DSCBehaviour, followed by the Strings "all-DS-IAT-processes", "all-DS-all-DS-IAT-processes", and `null`. The last argument here is null because of the special nature of the aggregate computation mode.

Recall that 'separate' mode computation command messages specified the meta-DAM containing the source DAMs whose statistics were to be separately computed. Aggregate mode operates similarly in that the meta-list is still specified, but rather than being collated, all results are collected together into one list. This list's name is taken directly from the third String. As this process results in only one list, no destination meta-list is required, and so the DStatComputer ignores this argument.

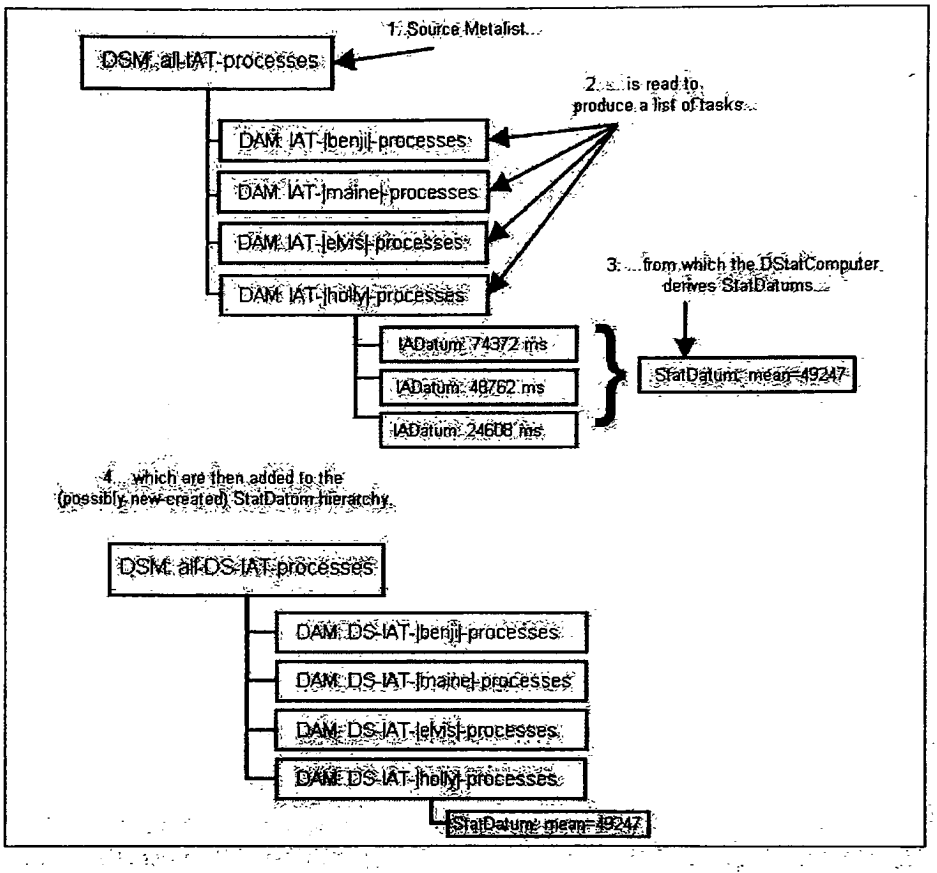


Figure 7.7: Statistic Computation, Phase 1

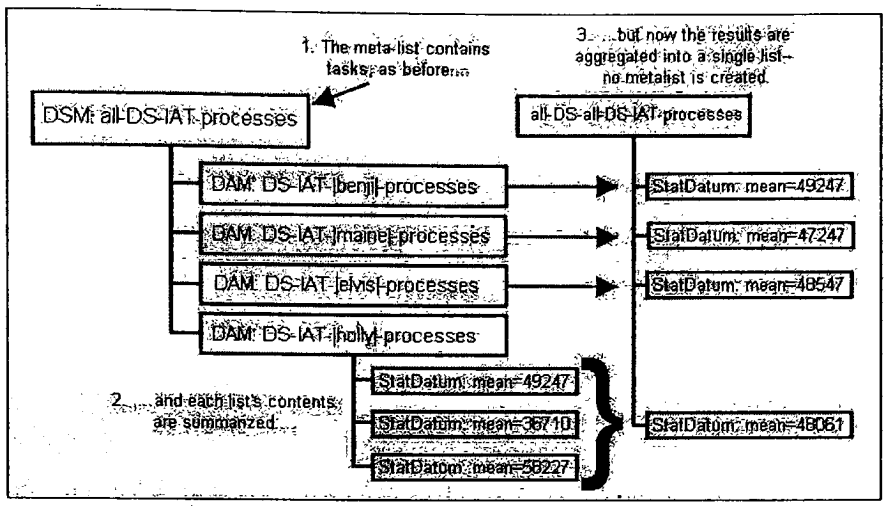


Figure 7.8: Statistic Computation, Phase 2

This process (see Figure 7.8) produces a DAM containing statistics for each $\overline{IAT_X}$. The statistics of the entire network can be computed from this DAM's contents: the ultimate bullet-point summary for the busy system administrator.

At this point, the user is faced with a pleasant embarassment of riches. The curious system administrator may wish to extract any of a variety of different overview statistics according to their specific operational requirements. For example, the admin may simply wish to know how long the 'average host' goes between visits. For that, they would use the computed average. Or, the admin may wish to monitor spikes in the hosts' IATs, in which case averages or maximums of host maximums would better meet their needs.

Phase 3: Network-wide statistics

Phase 2 produced or appended to a list of IAT means for the Net.Sense system. The transient nature of the data, coupled with their 'flat' storage arrangement, gives them the appearance of a sliding window of host statistics. This window facilitates examination of the recent Net.Sense system performance.

The testbed system is written to determine the average elapsed time between agent visits to hosts. As such, a third and final statistical computation is needed to determine the mean of all mean host IATs. In mathematical notation, the testbed system must compute $\overline{\overline{IAT_X}}$, the mean of the window described above.

This computation's process is simple by comparison with those of phases one and two. Source data are drawn from one list, and the resultant StatDatum is deposited in another, creating a sliding window similar to the product of phase 2 (See Figure 7.9). This obviates the need for complex source meta-list manipulation behaviours.

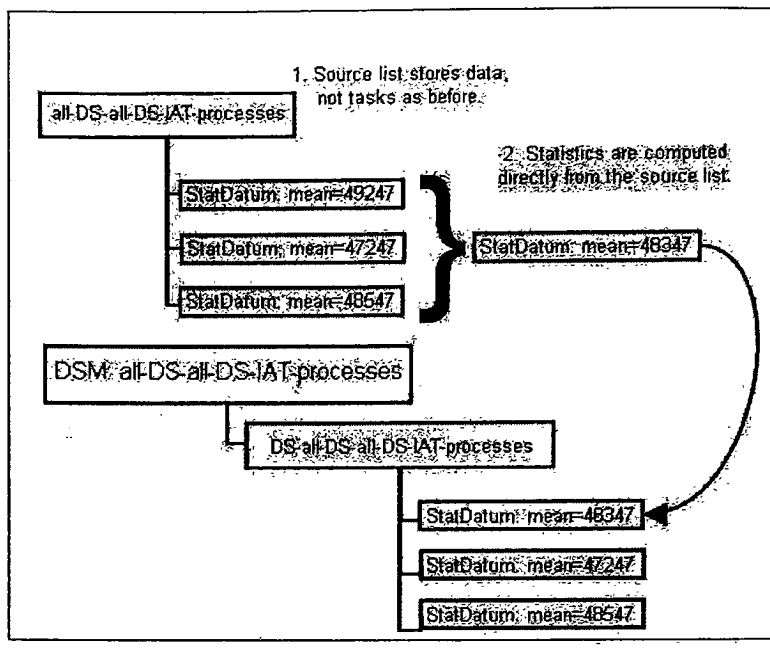


Figure 7.9: Statistic Computation, Phase 3

The phase three command message consists of the command mode constant (INDIVIDUAL), the source list 'all-DS-all-DS-IAT-processes', the destination list 'DS-all-DS-all-DS-IAT-processes', and the meta-destination list, 'all-DS-all-DS-all-DS-IAT-processes'. This command message updates or produces a list containing a sliding window of network-wide host IAT averages. These averages provide a temporal view of the average QoS provided by the Net.Sense system. As only one StatDatum is produced at a time, and as they are appended to the same list, the meta-list is not strictly needed. As the meta-list's presence does not materially affect the results, it was maintained to provide continuity of the testbed codebase during experimentation.

7.3 Review

This chapter has presented the testbed system. It has discussed in detail the three aspects of testbed operation (data collection, IAT computation, and statistic computation). In addition, the distributed data structures and their 'arrangement' within the JavaSpace used in each have been identified.

Part V

Experimentation and Results

Chapter 8

Experimentation

Time constraints precluded the execution of an experimental plan with the originally-planned depth and breadth. While many aspects of the Net.Sense prototype are of experimental interest, our formal experimental efforts are focussed on Hypothesis 1. To that end, over 110 hours of experimental run data was collected and analyzed.

Further, informal experimentation was conducted with respect to Hypothesis 2.1, and preliminary explorations of Hypothesis 2.2 were performed. This chapter discusses the performed experimentation to include the setup, execution, and subsequent analysis phases.

The discussion of these experiments follows.

8.1 Goal

Hypothesis 1 is paraphrased here for convenience. Recall that it states the primary factors in determining the provided QoS obey a relation of the form

$$aQ = nt$$

where

- n is the number of hosts in the managed network,
- a is the number of deployed sensory agents,
- t is the mean time an agent spends at a host, and
- Q is the QoS provided to the managed network.

Selection of independent variables was made with consideration to repeatability and consistency. As it is trivial to control n and a, and feasible to control t within a constant factor, Q was chosen as the dependent variable. Accordingly, Hypothesis 1 is recast into the form

$$Q = \frac{nt}{a}$$

where it is assumed that a, the number of sensory agents in the managed network, is greater than zero.

With the hypothesis rewritten in this way, the experimental design becomes clear. An experimental configuration point (CP) is an ordered pair (n, a). CPs are realized by initializing a Net.Sense system with the specified number of sensor agents on a network with the specified number of hosts. The goal of the experimental plan, then, is to perform experiments exercising the greatest number of practical CPs. programmed to move at the specified interval.

Observant readers will note the absence of t in the definition of a CP. This is a result of the t value being dependent on the agent loadout and other factors as described in Section 8.3.3.

The goal of the experimentation is to collect data for a variety of CPs. The goal of the analysis is to determine whether experimental data fit closely to a surface described by an equation of the rewritten hypothesis' form.

8.2 Plan

Clear goals provide for clear ideas. The experimental goal is to collect data on a variety of practical configurations. The first step in meeting this goal is to select configuration points (CPs) for testing. We identified the following CPs based on available resources. The notation used is (n, a), an ordered pair consisting of the sizes of the managed network and agent population, respectively.

(2,1)	(4,1)	(8,1)	(16,1)	(32,1)	(64,1)
		(8,2)	(16,2)	(32,2)	(64,2)
			(16,4)	(32,4)	(64,4)
				(32,8)	(64,8)
					(64,16)

A 'powers of two' scheme was used to select test points from the configuration space. The scheme was selected for its balance between space coverage and parsimony of time and computing resources. It provides good resolution for smaller network configurations while maximally exploiting available test resources. If CPs are viewed as ratios between n and a, this scheme provides a balance between ratio range (from 4 to 64) and ratio repetition (each ratio appears in between 1 and 5 CPs). (For more on the relevance of ratios, see Section 8.9.1.)

Experimentation consists of determining a QoS for each CP. The QoS metric chosen was a 95% confidence 90% upper DFTB (See Sections 3.3.1 and 4.2.1) on host mean IATs. The values for tolerance and confidence are arbitrary but are felt to be reasonable examples of system administrator needs. Assuming that swarm collection processes are under statistical control, users can be 95% confident that no more than 10% of the hosts' mean IATs (elements of $\overline{IAT_\chi}$) will exceed the QoS (DTFB) value.

8.3 Variable Identification

It is important to identify factors that may influence experimental results. This section lists factors whose effects were controlled, mitigated, or merely noted. Net.Sense systems are primarily affected by factors of three primary classes: internal, external, and configurational.

The class of internal variables includes such events as system failures resulting from faulty code, timing issues and race conditions. These events may have drastic effects on a running Net.Sense system, including data corruption, system state inconsistencies, or even agent death. These events can violate the integrity of nominally controlled variables, and as such tend to result in system shutdown and/or data invalidation. These events are effectively beyond control once code is frozen for experimentation.

External variables are factors that indirectly influence Net.Sense system operation. Examples include the selection of hosts in the managed network, the specifications of and load on those hosts, as well as the topology and load on the network connecting them. It is felt that disturbances in these variables will tend to be minor, having only slight effects on the Net.Sense system. However, more severe effects are possible and have in fact been observed during testing. These more severe effects include system crashes, network failures, and so on. Countermeasures employed to mitigate the effects of these variables are described below.

Configuration variables are (nearly) wholly controlled variables that experimenters manipulate when performing their experimentation. Presently, there are three such variables, as mentioned above. They are agent population (a), intermovement delay (t), and network size (n).

Having identified the classes of factors influencing an operational Net.Sense system, it is now feasible to discuss the factors and, where appropriate, their countermeasures.

8.3.1 Uncontrolled Factors

The effects of unrecoverable, unpredictable failures like those in this section were partially mitigated by operator observation and analysis of the resulting data set. When a failure of one of the above types was detected, the data was discarded.

Software Failures

Faults created by developer error may manifest in the form of system failure at any time during Net.Sense operation. They are effectively unpredictable, may result in arbitrarily catastrophic system failure, and are therefore considered the greatest threat to productive experimentation. Testing was performed, but continued operation has uncovered further software faults. After the code freeze for experimentation, however, no such faults could be repaired.

Hardware Failures

Triggered by a variety of causes, hardware issues can devestate the testbed system. Hardware failures (e.g. a disconnected power or network cable) occurring while an agent is operating on a host can destroy critical DDS (Section 6.5.5) as well as alter the agent population and host network size. These failures were simpler to detect, as they often impacted the managed network independently of Net.Sense.

8.3.2 Managed Factors

The effects of these unpredictable variabilities were partially mitigated through additions or modifications to the automated experimental procedure. Except where noted, these measures are felt to reasonably address the variance introduced by these factors.

System Load

To be of operational value, Net.Sense systems must coexist with the user and system software that justifies the existence of the managed network. Therefore, it was decided to perform experimentation on live, in-use systems. As as result, variation in system load was experienced depending on the time of day, day of the week, and week of the academic quarter.

Small, transient variations in system load should have no experimentally significant effects on the Net.Sense system, but more significant disturbances can cause significant effects. For example, runaway processes or heavy user jobs can consume inordinate amounts of RAM, CPU time, bandwidth, or process table entries. Like all processes, Net.Sense depends on these resources to function. Their lack can prevent the timely performance of Net.Sense activities including agent movement, InfoSpace manipulation, and other general agent tasks. As it is infeasible to continuously monitor the managed network, no useful account of system load is taken. However, it is felt that load effects are mitigated through the random selection of test network hosts. The random selection should lessen the risk that an occasional heavily- or lightly- loaded host will significantly skew experimental results.

Network Topology & Load

Arguments similar to those of the preceeding discussion apply to network load. The specific paths Net.Sense agents and data must trace through the network vary with the selection of hosts in the managed network. Further variability arises from conditions on both the traversed network segments and the routing/switching equipment that connects them. As no visibility into network usage, router load, etc. was available during experimentation, the author has deliberately neglected the effects of network load on the testbed system.

The impact of variable network topology have been mitigated by two means. The first and simplest of these means is the location of the InfoSpace (JavaSpace)

server on a subnet distinct from all other managed hosts. As such, all InfoSpace traffic must travel the 'worst case' path in the test network (through a minimum of two switches and a router). The second means is random selection of managed hosts across all available subnets. Both of these techniques are felt to reduce nonrepresentatively low IATs resulting from strictly local network traffic.

Host Specifications

The test network available for experimentation is a collection of Sun Ultra 10s. While mostly identical, some of these systems may have faster processors or more memory according to their purchase date and lab location. Since it is infeasible to change host configurations at the expense of operational readiness, random host selection was employed to prevent non-standard hosts from skewing experimental results.

8.3.3 Controlled Factors

Two of the following three variables are available for experimental manipulation. They are drawn directly from the restated form of Hypothesis 1 and the subsequent discussion of CPs. They are a (agent population), n (managed network size), and t (intermovement delay). These factors are referred to as 'controlled' because any changes in their values (and hence in the performance of the Net.Sense system) should be solely the result of the experimenter's configuration. As noted above, other events can effect these three factors, and so the reader should not assume that they are under perfect control.

Agent population a

The agent population a is simply the number of sensory agents operating within the managed network. It is felt that networks with larger numbers of agents will experience decreased agent IATs and hence a better (lower IAT) QoS. While a is primarily determined by the experimenter, events outside his/her control may alter this value as noted in the discussion above.

Managed Network Size n

The final controlled factor affecting Net.Sense operation is the size of the managed network, n. This factor determines the amount of work Net.Sense must perform to provide a given QoS. A larger managed network will require more visit events to maintain a given QoS, and may also require more agents.

Intermovement delay t

The intermovement delay t describes, on average, how much time an agent spends at a host. It is important because it determines the expected number of visits a swarm agent can make over a given time period. Thus, it is desirable for the value of t to be minimal.

In practice, the intermovement delay is more constrained than controlled. The minimum intermovement delay is currently set (hardcoded in `ns.infra.WanderBehavior`) at 20 seconds. The maximum intermovement delay, however, is potentially infinite but closely dependent on the tools and behaviours executing on an agent, along with their interactions with other agents. (Recall that Bugs do not move if they are executing tools or NSBs, and that DAMs locked for writing are inaccessible to other agents.) The value of t is further impacted by system loads and other factors as described in the previous section. It is assumed that the actual intermovement time value is generally limited to a small constant multiple of the 20 second base time.

8.4 Experimental Resources

As tested, Net.Sense managed networks consist of a server host and a number of client hosts. The server host provided Net.Sense infrastructure services (the DAMFactory, SingletonFactory, and the JavaSpace server), while client hosts are the systems which Net.Sense's primary purpose is to maintain. Both server and client hosts are part of the managed network and hence are serviced by the Net.Sense system.

Client hosts were drawn from a subset of the RIT Computer Science Department lab network that consists of approximately 70 Sun Ultra 10 workstations. These workstations are equipped with (various) 128 or 256MB of RAM, 100Mbps ethernet adapters, and UltraSparc processors. These systems run the Solaris 8 and 9 operating environments, with Sun's Java Development Kit (JDK) 1.4.0.

Our experiments used one of two machines for the server host. The machines are identical Sun Ultra 80 servers, each with 4GB RAM, 100Mbps ethernet, and 4 UltraSparc processors. This computing power far exceeded experimental requirements. Even under the most strenuous conditions, the JavaSpace memory footprint never exceeded 512MB, and informal observation indicated that server host CPU usage remained low. Nevertheless, machines used as server hosts in large networks should have copious amounts of RAM because the actual storage of the JavaSpace is not distributed across hosts. As a result, storing the InfoSpace can become a significant burden in large networks, especially when coupled with long Datum lifetimes.

8.5 Run Procedure

At present, the Net.Sense setup procedure is laborious, manual, and inflexible. This is largely due to time constraints imposed by the developmental issues discussed earlier. A more automated, customizable and user-friendly deployment procedure is slated for a future release.

8.5.1 Preconfiguration

This section describes the configuration and startup of a Net.Sense system. Changes to a small number of files must be made before testing on other than the RIT CS network. These files, with three exceptions, live in the `r` subdirectory off the main thesis directory.

- 'Run' scripts in the the `r` directory must be modified to suit the local experimental environment. In particular, the `container`, `space`, `monitor` and `tortureMonitor` scripts reference the server host by name, and must be modified accordingly. The `container`, `server`, `space`, `ssd` and `ssf` scripts must be modified to reflect the appropriate classpaths.
- Utility scripts and programs in the `tools` directory must be modified to reflect correct paths and classpaths.
- The `ns.infra.SpaceAccessor` and `ns.infra.TxnMgrAccessor` classes reference the server host by name. As a reminder, these classes will need to be recompiled after changes are made.
- The `build` script in the main thesis directory must be updated to reflect appropriate paths if experimenters wish to make use of it.

Once these steps are complete, the Net.Sense system is ready for startup.

8.5.2 Startup

The Net.Sense startup procedure requires four steps. In order of execution, they are:

1. Start the DAM and Singleton factories.

   This step is accomplished by executing the `r/ssd` and `r/ssf` scripts that launch the DAMFactory and SingletonFactory, respectively. The factories produce diagnostic output that the user may wish to capture. For convenience, it is recommended that these scripts be run in the foreground in different shell windows. While not required, it is recommended that these scripts be executed on the server host.

2. Start server processes.

   This step activates the JavaSpace and its supporting services, and launches the JADE platform with instructions to populate the swarm with the specified number of Bug agents. This task is accomplished by executing the `server` script on the server host with a positive integer command line argument indicating the number of swarm agents to spawn. For example, running `./server 8` on the server host would start the requisite services and spawn 8 sensory agents.

3. Populate the managed network.

   This step randomly selects hosts from a list and starts containers on them. These containers provide agents access to the hosts and make the hosts part of the managed network. To perform this task, execute the `tools/fireSlaves.pl` script with a positive integer command line argument indicating *the desired size of the managed network.* This script starts *one fewer* host than specified, because the server host (which is already running) is a host on the managed network. Hosts are randomly drawn from the `tools/longhostlist` file, ssh'd to, and instructed to launch a suitable JADE container.

4. Start a cleanup script, if desired.

   For experimental purposes, the testbed system logs collected process information to disk while operating (see Section 8.6). During continued operation, this can consume considerable disk space. Experimenters may wish to disable this functionality within the testbed system, or run a script that periodically removes `ProcessLister` output files from the run directory.

8.5.3 Operation

Once the Net.Sense testbed is operating, no further intervention is required. Interested experimenters may wish to explore the utilities provided in the `tools` directory, the data produced by the running Net.Sense system (deposited in the `run` directory), or any exceptions that may arise during operation. (Such exceptions are logged in files containing the word 'Problem' for recovered errors, and 'panic' for unhandled errors.) Experimenters wishing to monitor the QoS provided by the Net.Sense system are encouraged to make use of the IAT/DFTB analysis tools (see Section 8.7.1) throughout Net.Sense operation.

8.5.4 Shutdown & Archival

Shutdown and archival are presently manual processes. The shutdown process requires two steps. The first step is to shut down the JADE platform, and this can be accomplished in two ways. First, the experimenter may kill all processes associated with the JADE containers throughout the network. This process can be simplified by use of a script similar to that in `r/cleanup`, that serially SSH's to all potential hosts and kills any container processes.

The other, preferred method of shutting down the JADE system is through the `r/monitor` script that launches the main JADE GUI. The JADE containers can be terminated by clicking 'File' and then 'Shut down Agent Platform'.

Once the JADE system is deactivated, the experimenter must abort or kill the `server`, `ssd` and `ssf` scripts as well as any running cleanup scripts.

At this point, the Net.Sense system is wholly inactive, and collected data may be archived for later processing. The run data logged by the Net.Sense system will appear in a `run` subdirectory of the experimenter's home directory. The

archival process is accomplished by moving the 'run' directory into the correct subdirectory of the `archived-runs` directory. Subdirectories of 'archived-runs' are named with the pattern: *ncaa*. For example, run directories with CP (32, 8) should be renamed and moved into the `archived-runs/32c8a`.

8.6 Collection Procedure

Data collection is completely automated. Tools export their results to the JavaSpace and, optionally, to a text file in the run directory created at system startup. These files contain textual representations of the data elements being added to the JavaSpace at the completion of the tool's execution.

8.7 Postprocessing & Analysis

8.7.1 Run-based IAT-DFTB Analysis

Once raw run data is collected, it must be postprocessed into a form suitable for analysis. The first phase of postprocessing is local to each run, and is performed by the `tools/parseNetAvg.pl` and `tools/parseHostAvg.pl` scripts. The former extracts Phase 3 statistical data (See Section 7.2.3), while the latter extracts Phase 2 data (See Section 7.2.3). Rather than executing these scripts directly, it is likely that the experimenter is more interested in executing the `tools/showNetStats` and `showAllHostStats` scripts. The latter two scripts call the former two and subsequently produce Postscript graphs of their results in a file called `plot.ps`.

Note that at the present time, the `tools/parseNetAvg.pl` and `tools/showNetStats` scripts are not used. Meeting the experimental plan's goals called for shorter (1-2 hour) runs, which did not provide enough time for Phase 3 results to come under statistical control.

Run postprocessing begins with extraction of the relevant data from exported log files. The 'relevant data' are the date, Datum tag, remaining lease time, datum count, and the value of StatDatum's 'mean' field. These data are extracted, de-duplicated, and sorted into two files ('temporal' and 'monoStat') by time and mean fields. The list of mean field values is fed to a Java program (`tools.OSTB`) that computes the desired DFTB. This value is used to define a function with the (constant) DFTB value.

The master GNUplot file (`tools/masterStatPlot.gnu`) displays the collected data in two series. The first is ordered by time, and provides a means to visually verify that the data are under statistical control. The second is ordered by value, and provides a 'rule of thumb' for estimating quartiles and identifying the modality of the collected data. Finally, GNUplot overlays the DFTB function from the previous step, allowing the experimenter to visually assess the DFTB's validity and distance from the data's mode(s). See Figure 8.1 for an example graph.

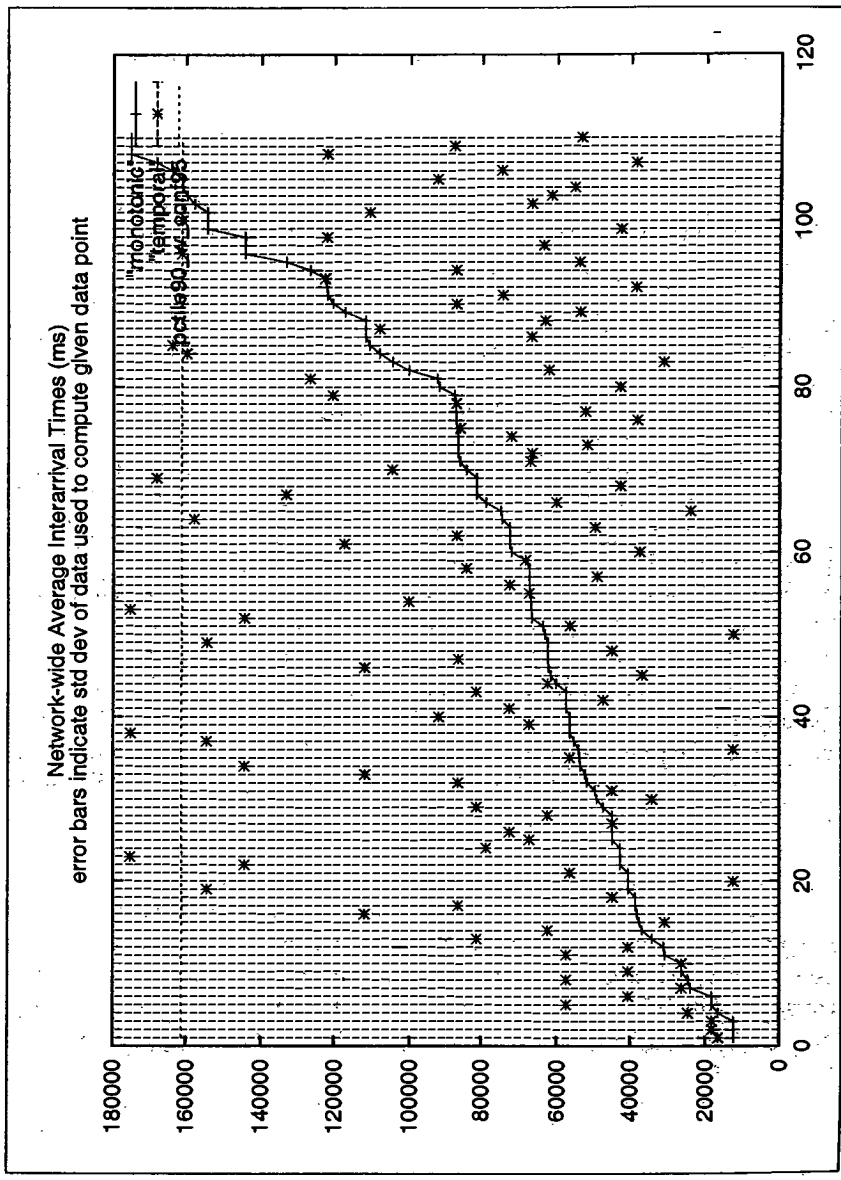


Figure 8.1: IAT / DFTB plot (16,4), run 2

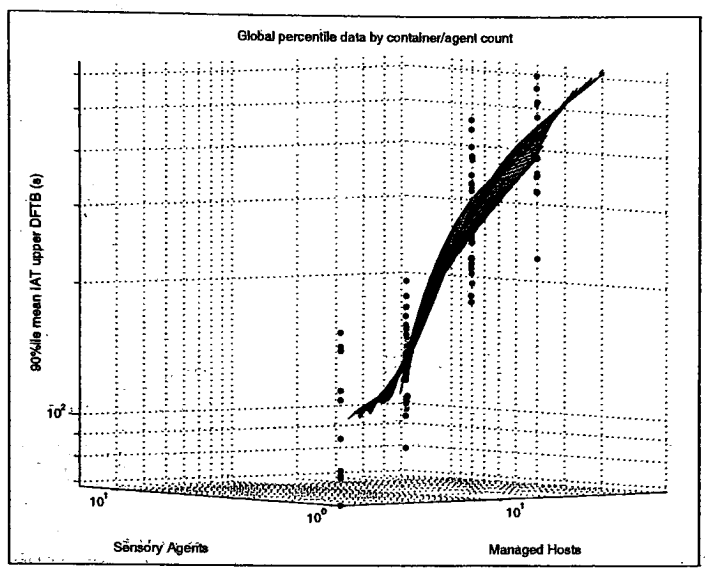


Figure 8.2: QoS DFTBs by Configuration Point (View 1)

8.7.2 All-CP Analysis

DFTBs resulting from the above process are collected into a list by the `tools/extractCutoffs` script. This process is performed for each archive of experimental data. The resulting tabular files are then manually merged and imported into Matlab for display and further analysis.

In order to assess the veracity of Hypothesis 1, the collected DFTBs are grouped and plotted by CP. Each CP group's values are averaged, and a surface is cubically interpolated between the resultant means using a custom Matlab function in `globalPct.m`, located in Appendix 9.2.3. Figures 8.2 and 8.3 provide two views of such a plot.

To evaluate the hypothesis' validity, equations of the revised hypothetical form were evaluated and plotted on a grid of evenly spaced pseudo-CPs by a custom Matlab function in `straightRel.m`, located in Appendix 9.2.3. Manual estimation and visual inspection determined coefficients that bracket the collected QoS data with hypothetical surfaces.

In Figure 8.4, three planes overlay the CP group mean surface. Each plane is the equation of the revised hypothetical form multiplied by the one of the three determined coefficients. Depicted are coefficients with value 45 (red), 35 (green), and 25 (blue). Note the way in which the red and blue planes tightly bracket the surface described by the CP group means. The tightness of the bracketing planes to the data surface indicates that the data surface points are interrelated in a manner similar to that of the bounding planes' points.

The final analysis performed on the collected data was a multivariate least-

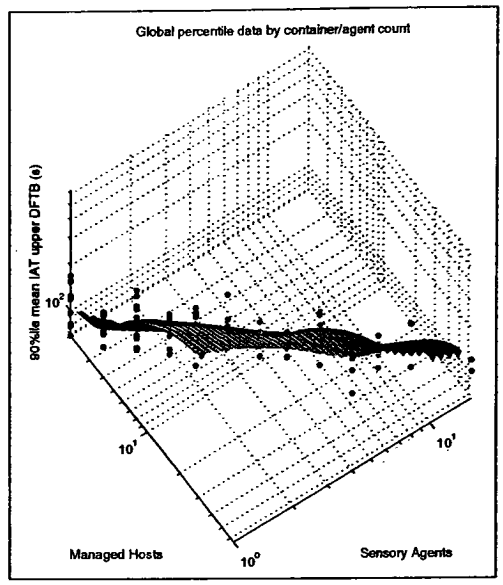


Figure 8.3: QoS DFTBs by Configuration Point (View 2)

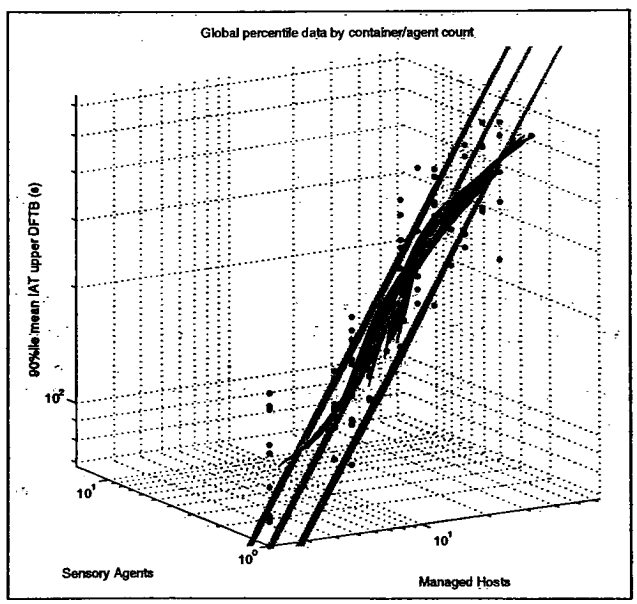


Figure 8.4: QoS DFTBs by CP with 3 constant-coefficient Hypothesis 1 planes

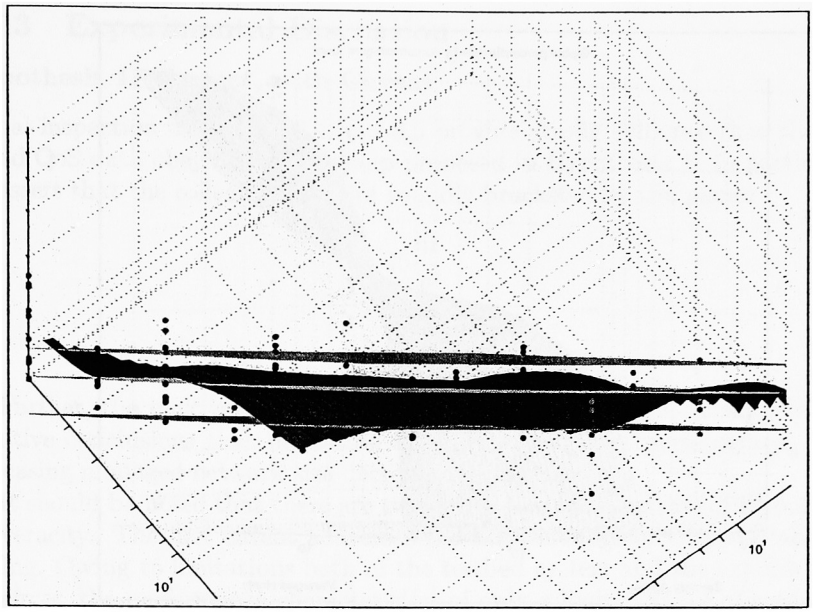


Figure 8.5: QoS DFTBs by CP with3 constant-coefficient Hypothesis 1 planes (Top View, close zoom)

squares regression. This was performed by a custom Matlab function in `MVRegress.m` that can be found in Appendix 9.2.3. The regression attempted to identify 2 coefficients X and Y such that the total distance between the surface defined by

$$Z + \frac{Xnt}{Ya}$$

is minimized. This regression yielded coefficients $Z = 247.7436$, $X = 7.0850$, and $Y = -41.7646$.

These coefficients illustrate the impact of individual variables on Net.Sense performance. $X = 7.0850$ indicates positive correlation between increased values of n and QoS DFTBs. In other words, adding hosts to the managed network lengthens the window within which users can be confident that hosts will be visited. $Y = -41.7646$ indicates negative correlation between increased values of a and QoS DFTBs. In other words, augmenting the swarm shortens the window within which users can be confident that hosts will be visited. It is felt that further analysis of the data may uncover additional interesting trends.

Two views of the resultant regression surface are shown in Figures 8.6 and 8.7.

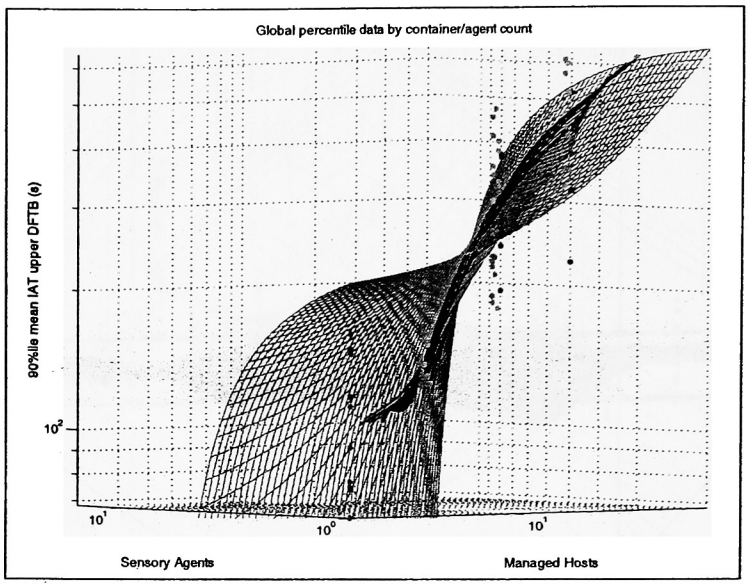


Figure 8.6: QoS DFTBs by CP with multivariate regression surface (View 1)

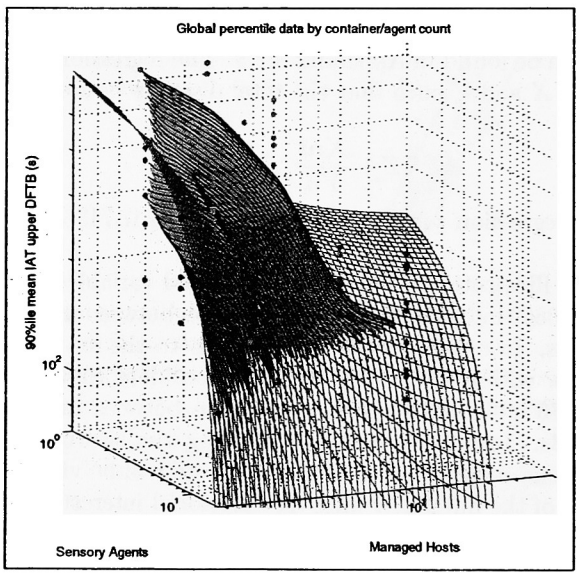


Figure 8.7: QoS DFTBs by CP with multivariate regression surface (View 2)

8.7.3 Experimental Discussion

Hypothesis 1: Correct, with Caveat

Visual inspection of Figures 8.4 and 8.5 provide strong evidence that the collected QoS data obey the relationship proposed in Hypothesis 1. In particular, we assert that the collected data are soundly bracketed by the planes

$$45\frac{nt}{a}$$

and

$$25\frac{nt}{a}$$

Further, the multivariate regression discussed in Section 8.7.2 supports the intuitive conclusions that increasing swarm size increases performance, while increasing managed network size decreases performance.

It should be noted that there are two caveats on the assertion of Hypothesis 1's veracity. The first caveat addresses the limited range of CPs available for testing. Owing to limitations both of the testbed system and the experimental resources, the performance characteristics of only a limited range of configuration points were able to be explored.

The second caveat addresses the quality of the collected data. The author believes that reasonable measures were taken to ensure the overall validity of the experimental methods. It is believed that significant opportunities exist to improve the tightness of the true (asymptotic) mean CP QoS values. For example, much extended experiment durations would enable the use of distributed free tolerance intervals, rather than upper bounds. This would enable the experimenter to observe QoS values closer to the true asymptotic values. Additional measures could be taken to further reduce the impact of uncontrolled factors upon experimental results. In other words, the experimental processes can be further refined, and the author recognizes that such refinements may uncover new and more accurate relations than that of Hypothesis 1.

Based on the experimental method and the data collected, however, we are confident asserting that, for the tested CPs, the QoS provided by Net.Sense is bounded by a relation in the revised form of Hypothesis 1.

8.8 Data Summary

This section contains the collected CP data, sorted by container and agent count. The following table provides the sorted DFTB data used in the previous analysis for experimenters wishing to perform their own analyses or investigations. (Note: values of Q are in seconds.)

n	a	Q	n	a	Q	n	a	Q	n	a	Q
2	1	119.4031	2	1	125.6130	2	1	171.0470	2	1	78.9837
2	1	79.7831	2	1	82.0420	2	1	97.3026	2	1	125.6130
2	1	155.3581	2	1	158.7928	2	1	68.3296	2	1	78.9837
2	1	79.7831	2	1	82.0420	2	1	97.3026	4	1	108.5875
4	1	132.6673	4	1	146.2952	4	1	149.1314	4	1	151.6201
4	1	176.2230	4	1	108.5875	4	1	131.5300	4	1	137.4252
4	1	138.4097	4	1	141.2030	4	1	141.2442	4	1	151.7020
4	1	184.6770	8	1	195.9945	8	1	245.8003	8	1	278.3383
8	1	323.5147	8	1	366.2950	8	1	487.6828	8	1	202.5017
8	1	233.9493	8	1	239.7970	8	1	275.2701	8	1	278.4996
8	1	383.0913	8	1	445.5148	8	2	112.0746	8	2	114.9898
8	2	142.1939	8	2	165.8860	8	2	197.5426	8	2	88.2629
8	2	114.3974	8	2	114.6096	8	2	115.3878	8	2	133.2907
8	2	142.1939	8	2	161.2524	8	2	186.5750	8	2	213.6914
16	1	374.4980	16	1	345.9890	16	1	458.4148	16	1	599.3386
16	1	642.1466	16	2	275.5855	16	2	306.2563	16	2	219.3785
16	2	238.4446	16	2	258.1151	16	2	273.7755	16	2	338.8369
16	2	495.8095	16	4	110.2795	16	4	125.7814	16	4	153.3603
16	4	160.9947	16	4	108.9229	16	4	130.2225	16	4	130.9616
16	4	139.9493	16	4	146.0454	16	4	161.0961	32	1	640.4128
32	2	361.0470	32	2	363.6703	32	2	367.7280	32	2	424.8960
32	2	433.4010	32	2	446.3800	32	2	531.4852	32	2	615.6850
32	4	181.5908	32	4	290.1255	32	4	290.1312	32	4	292.9120
32	4	293.3780	32	4	311.3750	32	4	331.7890	32	4	336.8480
32	4	393.2796	32	4	412.3130	32	8	164.2504	32	8	100.7481
32	8	122.6858	32	8	145.7029	64	4	224.8245	64	4	319.8780
64	4	382.6450	64	4	477.4480	64	4	517.2630	64	8	193.0820
64	8	240.1768	64	8	258.3216	64	8	371.6670	64	8	376.5390
64	16	103.8366	64	16	123.1953						

8.9 Experimental Notes / Observations

This section collects some informal observations made during experimentation and testbed use.

8.9.1 Ratios

CPs can be thought of as representing a ratio between the network size n and the agent count a. Intuitively, infinitely many CPs represent a given ratio. Interesting data could potentially be cleaned by looking for 'iso-QoS' bands whose constituent points share ratios, or by searching for scalability information by comparing the group means of same-ratio CPs. Significant differences in QoS among same-ratio CPs could be indicative of future scalability issues.

8.9.2 64-container experimentation

Little experimentation was performed with managed networks of 64 hosts. The decision to cease experimentation at $(64, *)$ CPs was based on three factors. First, an alarming number of DDS access collisions dramatically reduced the volume of data extracted from the managed network. As such, we have lower confidence in the validity of results from this CP series. It appears that access collisions must be reduced before $(64, *)$ experimentation can proceed in earnest. Such a reduction may be obtainable via further refinements to DDS write methods, or by deliberately lowering agent count, thereby reducing contention for DDS resources.

8.9.3 InfoSpace storage requirements

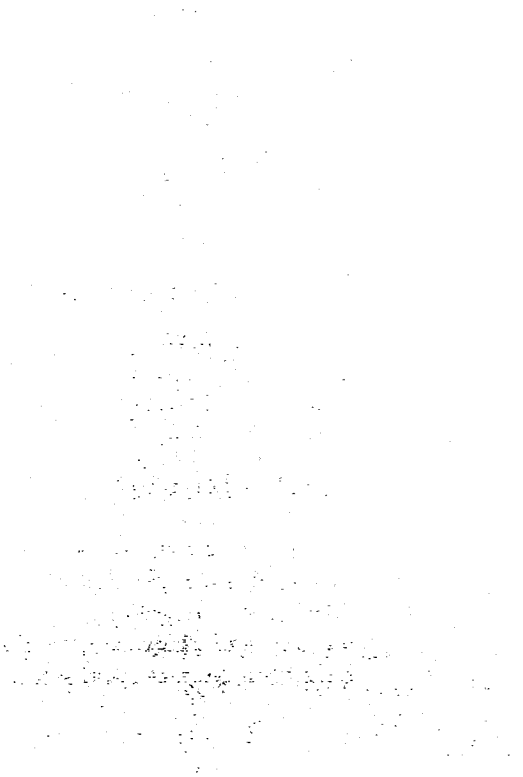
It is felt that machines selected for server hosts should have copious amounts of RAM. JavaSpaces provide location transparency for access, but do not distribute storage requirements across hosts. As a result, the InfoSpace can become a significant burden in large networks, especially when coupled with long Datum lifetimes. In our experimentation, a maximum size of 512MB was never exceeded.

8.9.4 Multiple archive repositories

If experimenters wish to maintain multiple archived run repositories, they will need to either alter the `tools/extractCutoffs` script or manually rename each repository to 'archived-runs' and reexecute the script.

8.10 Review

This chapter has discussed the processes and scripts used to perform experiments with the Net.Sense testbed system, including testbed startup and shutdown, data collections, and post-processing. Collected data have been summarized and analyzed with discussion, and miscellaneous informal observations have been collected and summarized.



Chapter 9

Conclusions and Future Work

This chapter discusses conclusions and future work required to bring the prototype framework and testbed system closer to operational readiness and the realization of the Net.Sense vision.

9.1 Conclusions

This thesis has presented the current state of the Net.Sense vision. To demonstrate the vision's principles, a prototype framework was developed and used to construct a testbed system. This testbed assisted the exploration of various design questions and helped crystallize the workflow required to perform the functions outlined in the vision. In addition, the testbed provided a means to a preliminary characterization of Net.Sense performance characteristics.

9.1.1 Research Effort Review

This thesis has significantly advanced the state of the Net.Sense vision. At inception, this vision was well-developed in some areas, underdeveloped or erroneous in others. Framework and testbed development prompted further expansion of the Net.Sense vision, adding detail to existing areas of the vision as well as demonstrating a need for wholly new areas. For example, it was found that the vision as written provided inadequate depth in its discussion of tool-behavior interactions, and failed to address the categories and formats of structures within the InfoSpace.

Before design and development could proceed, it was necessary to procure the technical underpinnings of the prototype. As the focus of this thesis is not the implementation of these underpinnings, the available technologies were surveyed. These surveys led to the selection of JADE as the base agent framework and JavaSpaces as the foundation for the InfoSpace. Once selected, exploratory

development projects were performed to familiarize ourselves with the intricacies of the selected tools. The WanderBehavior class, which drives the testbed system's sensory agents, evolved from such an exploration.

Once familiar with the selected technologies, the author began a cyclically interleaved design and development process. The interleaving reduced delay between the development of a framework technology and the testbed code that utilizes it. This decision proved to be instrumental to the development effort's successes as it uncovered significant functionality and usability issues at an early stage. Nevertheless, inadequacies discovered in the original Net.Sense vision proved to be a significant obstacle to development. The lesson learned is that designing without detailed requirements is as dangerous as coding without a detailed design.

When the prototype framework was nominally complete, development effort focused on transforming the development validation platform into the experimental testbed found in the stat package. This transformation was not without its growing pains. Several limitations of the prototype InfoSpace were unearthed, necessitating extensions to the DDS family as well as modifications to the behavior system that manipulates them. The IAT computation process posed particular challenges as it required maintenance of state (MRPDs) related to but distinct from the DAMs being processed.

The statistic computation process raised further issues. The complexities of intricate dataset manipulations are compounded by a desire for tool generality, leading to overcomplicated tools that rely on extremely complex subbehaviours. Our experiences with the testbed's DStatComputer lead us to believe that data-processing tools in the Net.Sense system should be viewed not as processes but as data transforms or filters. This perspective would formally separate the tasks of acquiring and processing data. It is felt that this separation would greatly simplify tool design.

Having brought both framework and testbed to a functional state, the question of feasibility became one of efficiency. How well would the Net.Sense system perform? Intuition and prior simulation experience led to the proposition of Hypotheses 1 and 2. However, due to time constraints, only Hypothesis 1 was rigorously explored. An experimental plan for Hypothesis 1 was developed and executed. Over the course of the execution approximately 114 hours of experimental data comprising were collected.

These data were analyzed at both run and global scopes. Analysis of individual run data provided visual indications of Net.Sense's microscale operating characteristics, such as IAT distribution. Subsequent postprocessing created a 3-D surface representative of the global aggregation of run data. This surface provides an overview of the testbed's performance across the tested CPs.

Surfaces were generated by equations of Hypothesis 1's revised form and visually compared with the aggregate data surface. The comparison indicated that modulo a constant coefficient, the Hypothesis 1 relation adequately described testbed system performance. Subsequent multivariate regression revealed additional details of the control variables' impact on testbed performance. Specifically, they indicated that adding a host to the managed network increases QoS

values (worsening performance), while adding agents to the swarm decreases QoS values (improving performance).

With this initial validation of the primary hypothesis, it remained only to note certain informal experimental observations and establish directions for future efforts. The first of these future directions must necessarily be the elimination of known critical code faults in the prototype system. These faults can affect system stability and data integrity, and so must be addressed as a top priority.

Once all critical development issues are addressed, usability, scalability, and functionality enhancements will be considered. Present write-locking algorithms are known to limit prototype scalability. These limitations preclude consistent, reliable testbed operation for CPs with large numbers of agents. As a result, they must be addressed before experimentation can continue. Further, operational conveniences are also planned. These include a more flexible configuration system, and control script that enable the batch, unattended execution of a given test plan.

9.2 Future Work

Both the prototype framework and the testbed system require significant future work before being deemed ready for production use. A short list of these items, with brief discussions, are provided both to raise awareness of issues experimenters may encounter with the Net.Sense prototype, as well as to lay out avenues of work for future releases.

9.2.1 Prototype Framework

The prototype framework issues fall into three categories: existing design/implementation faults, performance/scalability optimizations and enhancements to more fully realize the Net.Sense vision.

Infrastructure Services

Implementation Fault: Centralized InfoSpace

> The present InfoSpace implementation relies on a single, monolithic JavaSpace. While suitable for a proof of concept, this implementation creates a particularly critical single point of failure within the Net.Sense system. Future implementations should look to alternative systems which provide load balancing and enhanced availability through replication. (See Section 8.4)

Optimization: JavaSpace 'Queries'

> Presently, JavaSpaces offer mechanisms for manipulating single elements within a JavaSpace through its read(), take(), and put() methods.

Net.Sense system function would be measurably enhanced by two features: JavaSpace 'queries' similar to database queries, can return multiple results, and storage distribution, to balance the memory requirements across a network of agents or hosts. (See Section 6.5.3)

Vision Extension, Optimization: DDS Factories

At present, DDSs are created by unique, standalone factories living outside the agent community. Making this functionality a service of an agent subcommunity would not only advance the Net.Sense vision, but could potentially provide significant performance and scalability benefits. (See Sections 6.5.3 and 6.6.2)

Feature Enhancement: DDS Factories

At present, mechanisms are provided for neither properly deleting elements from the InfoSpace nor on-demand InfoSpace garbage collection. These features (especially the former) can be of value in systems where host stability or connectivity is a significant issue.

Design Fault: DDS DoS 6.5.5

At present, poorly constructed or malicious behaviors can 'steal' DDSs out of the InfoSpace. While only a nuisance when speaking of transient DDS, the issue becomes critical when applied to permanent DDS, as factories will refuse to recreate the missing DDS. This issue requires careful consideration, and potentially significant modifications to both the DDS and factory codebases. As such, its resolution is deferred to a later release. (See Section 6.5.5)

Implementation Fault: DAM Garbage Collection

A fault in the DAMFactory garbage collection routine can result in data loss if a DAM is garbage-collected while locked by an agent. Future implementations will resolve this issue, perhaps via a 'check-out' or leasing mechanism. (See Section 6.5.3)

Distributed Data Structures

Design Fault: DDS Names

As the `ns.DAM` behaviours evolved, APIs, naming conventions, and entire new class hierarchies evolved with them. As a result, DDS class names are sometimes inconsistent or confusing. These naming issues will be resolved in the next release. (See Section 6.1.1 and 6.2.1)

Optimization: DDS Write performance

Informal observations indicate that Configuration points with large a fail to scale as well as would be hoped. Informal log analysis suggests that the issue centers around resource contention issues brought on by write-locking inefficiencies. Potential solutions (e.g. replacing the JavaSpace

with a proper distributed database or decreasing lock granularity) exist, but must be explored prior to implementation. (See Section 6.5.3)

Design Fault: DDS Multiple Write

Malicious or miswritten behaviours that access the JavaSpace directly could perform multiple unlock() calls, thereby writing multiple instances of the element in question into the InfoSpace, and violating the uniqueness property of Singletons and permanent DAMs. Because JavaSpace interactions are only intended to be performed in pairs by the provided DAM- and SingletonOpBehaviours, no special safeguards exist to entry mishandling as described above. The author foresees this issue requiring significant effort to be addressed in a suitably robust fashion. (See Section 6.2.1)

Design Fault: DAM Nesting Logic

Ad-hoc logic in the DAM class provides rudimentary protection against double-adding 'special-case creation' objects such as DAMs and Singletons to the InfoSpace. While effective, the present implementation does not lend itself to further infrastructure development. More flexible implementations (e.g. marker interfaces) are planned for use in future releases. (See Section 6.5.3)

Feature Enhancement: Complex Value Extraction 7.1.2

It may be reasonable to view 'complex' data elements such as StatDatums in a variety of ways. Consider Phase 2 and 3 statistics computation in the testbed system. When asked for its value, StatDatums return their means. As a result, statistics computed on StatDatums are statistics on the mean fields of the source StatDatum objects. The current implementation does not provide flexibility to select other fields of the StatDatum for computation. To provide this flexibility, we propose user-defined filters which transform a List of StatDatums into a List of Double objects containing the desired StatDatum field.

In general use, these filters would be applied to the DAM or List of source Datums to be processed. The resulting data would be suitable as input to a tool such as a DStatComputer.

Tools

Feature Enhancement: InfoSpace manipulation behaviours

As noted in Section 6.1.1, the existing tool result publication process requires refactoring and standardization. It is believed that a suitably general implementation should eliminate a proportionately large amount of code from simple tools such as sensors.

NS Behaviour System

Implementation Fault: Subbehaviour Access Controls

Known issues with the GenericOpBehavior (See Section 6.2.1) permit subclasses direct access to the phase behaviour objects stored within the GenericOpBehaviour, allowing them to be changed at any time. As a result, careless subclasses could change these behaviours once execution is underway, potentially corrupting the computation in progress and its associated data structures. in the next release.

Feature Enhancement: NSB Stack Trace

While debugging, the author has derived significant value from the existing MB/MBMB exception handling mechanism. (See Section 6.4.2) However, it is felt that a behavioural 'stack trace' indicating which behaviours led to the exception and where the exception occured would be very useful.

Design Fault: runThrough/write-locking

While the resource locking issue mentioned in Sections 6.4.2 and 9.2.1 has a partial workaround in the run-through mechanism, it is unclear that this mechanism is the correct solution. Investigation into multithreading is merited before investing further development effort into the many and various issues arising from the prototype framework's interactions with the JADE scheduler. It is felt that a 'task' framework implemented in the style of virtual threads (potentially layered atop real threads) may help address both the resource locking and run-through deadlock issues.

Feature Enhancement: Security Model

The prototype framework and the testbed system that utilizes it have no concept of security. Information is freely available to agents and non-agents alike; no assurances about DDS integrity are provided, and the InfoSpace can be trivially DoS'd. These issues must be addressed before Net.Sense can be considered operationally ready.

Feature Enhancement: Dynamic class (un)loading

The Net.Sense vision calls for dynamic loading and unloading of both tools and behaviors. (See Section 6.6.1) That portion of the vision poses nontrivial security issues on top of the already significant cataloguing, versioning, distribution, and results-management issues. In addition to the logistical issues, work is required to correctly address the issues raised by stopping, starting and (un)registering behaviors and their provided services.

Implementation Fault, Feature Enhancement: NSB control logic issues

The prototype framework does not correctly handle the tasks of managing NSB activation permissions associated with agent movement safety. These faults could allow an agent to move while an NSB or tool is still active, potentially causing data loss. (See Section 6.6.3)

Agent Internals

Feature Enhancement: Behaviour Signalling / Message Channels

The current, broadcast communication mechanism becomes inefficient when primarily limited-destination messages are received by agents with many behaviors. (See Section 6.3) It is likely that future implementations will address these issues through a mailboxing system or other, similar multichannel communication mechanism that supports agent-component-level addressing (See Section 6.6.2).

Design Fault: Multithreading

As mentioned elsewhere (See Sections 6.3, and 6.6.1), significant development challenges were encountered while attempting to operate within the constraints of the JADE behaviour and scheduling system. Thus far, the most promising solution is a form of thread-based task framework that extends Java threading concepts into the multiagent domain. Careful consideration must be given to those functions (such as agent movement) which cannot be threaded and hence exist 'underneath' the proposed tasking model.

Feature Enhancement. Exploit Ontologies

Ontologies are currently only utilized in limited, intra-agent event notifications (See Section 6.1.1). Automated reasoning capabilities and agent interoperability in large, diverse Net.Sense systems will demand *lingua(s) franca.*

Design Fault: MB Retry-Count

Currently, retry counts are determined internally by the user MB, while the interattempt delay is determined by the BaseMB. (See Section 6.4.3) It is felt that usability would be enhanced by giving both pieces of state to one or the other of the two classes.

Design Fault: Tool Access Control

Currently, NSBs request Tools by name, and Toolbelts return references to the requested tool, if it is found. This poses security and reentrancy issues. It is felt that these issues could best be solved by returning a form of tool proxy to the client NSB. In addition to increasing system security, this solution could dovetail nicely with the task model discussed earlier by providing 'run-once' control for tool accesses.

9.2.2 Testbed System

Feature Enhancement: Expert System Integration

The Net.Sense vision calls for agents with reasoning capabilities to monitor and maintain system states. As noted in Section 6.5.2, the prototype implementation of such a closed-loop system is presently inoperable due

to changes in the InfoSpace implementation. The necessary corrections must be made to the overwatch behaviours to continue down the path towards integrated reasoning capability, for example as it relates to swarm population manangement. (See Section 4.2.2)

Feature Enhancement: User Interface

Starting and stopping the Net.Sense testbed system is a complicated and potentially error-prone process. A command-line tool that provided a single control interface for experimental configuration, initiation and termination would make present and future testbed systems more accessible to researchers and users alike.

Feature Enhancement: Expanded Visualisation Capabilities

The tools currently provided for introspecting the JavaSpace of an operating Net.Sense system are nonintuitive. A GUI browser with hierarchical trees or menus reflecting the DDS hierarchies within the JavaSpace would be useful both for debugging and demonstration purposes.

9.2.3 General

Design Fault: Misplaced Classes

Several classes in the `stat` package have been sufficiently generalized to merit inclusion in the Net.Sense framework proper (see Section 6.1.2). These classes will be migrated to a new `ns.stat` package while the remainder move to a strictly testbed package.

Implementation Fault: Misnamed ListenBehavior

The ListenBehavior is not an NSBehavior but an instance of a JADE behaviour (See Section 6.6.2). Therefore, it will be renamed to `ListenBehaivour` to conform to the naming convention.

Implementation Fault: Configuration Inflexible

The process of starting up and shutting down the testbed system is significantly accelerated if ssh key agents or a similar 'hands-off' authentication methodology is employed. This is especially true in larger test networks, where manually reentering passwords for 64 or 128 machines would be prohibitive.

Feature Enhancement: Automated Test Harness

Periodically, experimenters must perform IAT/DFTB analysis on runs in progress to determine whether the run has collected enough data to analyze. This adds an element of variability to the experimentation process through the experimenter. A preferable solution would be to automate startup and shutdown processes, and implement a trigger based on automated IAT/DFTB run analysis. In this way, runs could be entirely

scripted, permitting the automated test plan execution and eliminating any human interference in the experimental process.

Appendix A

Supplementary Code

This chapter contains scripts, functions, programs, and routines that are related to but not part of the main Net.Sense system. These utilities test aspects of the prototype framework, automate testbed startup and assist in the performance of run postprocessing and analysis.

A.1 Testbed Control

This section introduces the scripts which help control the testbed system. Many of these scripts make use to one or more of a series of aliases. These aliases ('icl1', 'icl2', 'icl3', 'icl4', 'icl5', 'csl' and 'gradlab') expand to a for loop which iterates a variable `$i` over the list of hostnames for the eponymous lab.

Other commonly-used aliases include:

- `nuke = '/bin/rm -r -f'`
- `blast = '/bin/rm -f'`

A.1.1 Startup Scripts

Scripts in this section simplify the process of bringing a testbed system online.

`r/server`

Experimenters execute `r/server` to start a server on the current host. It is a wrapper for the `fireServer.pl` script discussed below.

```
#!/usr/local/bin/zsh

if ([[ -e /run ]]); then
  print CAUTION: existing data found.
  print Please move away the run directory.
  exit
```

```
else
  ~/thesis/r/space
  sleep 4
  `perl ~/thesis/tools/fireServer.pl $1`
fi
```

tools/fireServer.pl

The tools/fireServer.pl script, when given the desired size of the swarm, prints a command line string which will activate the server with the requested number of swarm agents as well as one DStatAgent and one IATAgent.

```
#!/usr/local/bin/perl


use strict;
use IO::File;

die "Usage: fireServer.pl #bugs" unless @ARGV > 0;
my($initString) = "java -Djava.security.policy=" .
  "/home/stu10/s18/jjm7570/jini1_2_1/policy/policy.all ";
$initString = $initString .
  "jade.Boot -port 20029 iBug:stat.IATAgent dsBug:stat.
      DStatAgent";

foreach (1..$ARGV[0]) {
  $initString = $initString . " Bug$_:stat.Bug";
}

print "$initString \n";
```

r/space

The r/space script prepares for a new experimental run by cleaning up any files remaining from the prior run and then starting a new JavaSpace and its supporting RMID and RMI Registry services.

```
#!/usr/local/bin/zsh

#to start the RMI Daemon
nuke ~/reggielog
nuke ~/log
nuke ~/txnlog
nuke ~/serverLog
mkdir ~/run

rmid -J-Dsun.rmi.activation.execPolicy=none \
   -J-Djava.security.policy=jini1_2_1/lib/rmid.policy &
```

```
echo "starting reggie"

#to start reggie
java -jar jini1_2_1/lib/reggie.jar\
    http://www.cs.rit.edu/~jjm7570/thesis/reggie-dl.jar \
    jini1_2_1/policy/policy.all reggielog public

echo "starting javaspace"
#to start the javaspace
java -jar -Xmx512m -Djava.security.policy=\
     jini1_2_1/policy/policy.all \
     -Djava.rmi.server.codebase=\
       http://www.cs.rit.edu/~jjm7570/thesis/outrigger-dl.jar
           \
     jini1_2_1/lib/transient-outrigger jar \
     jini://hilly.cs.rit.edu &
```

ssd

The ssd script initializes a DAMFactory. Recall that the DAMFactory is a standalone infrastructure server which extracts DAM creation requests from the JavaSpace. If the requested DAM does not already exist, the factory creates it.

```
#!/usr/local/bin/zsh

java -Djava.security.policy=jini1_2_1/policy/policy.all \
  -Djava.rmi.server.codebase=\
     http://www.cs.rit.edu/~jjm7570/thesis/ \
  ns.dam.DAMFactory
```

ssf

The ssd script initializes a SingletonFactory. Recall that the SingletonFactory is a standalone infrastructure server which extracts Singleton creation requests from the JavaSpace. If the requested Singleton does not already exist, the factory creates it.

```
#!/usr/local/bin/zsh

java -Djava.security.policy=jini1_2_1/policy/policy.all \
  -Djava.rmi.server.codebase=\
     http://www.cs.rit.edu/~jjm7570/thesis/ \
  ns.dam.DAMFactory
```

container

The container script initializes creates a JADE container on the local machine. This container connects back to the main platform host created indirectly by the server script.

```
#!/usr/local/bin/zsh

java -Djava.security.policy=jini1_2_1/policy/policy.all \
  -Djava.rmi.server.codebase=\
    http://www.cs.rit.edu/~jjm7570/thesis/ \
  ns.dam.DAMFactory
```

A.1.2 tools/fireSlaves.pl

The tools/fireSlaves.pl is invoked by the user at testbed startup, to randomly select and start containers on one fewer than the requested number of hosts. Recall that one fewer host is started because the server hosting the main JADE platform container is itself a host.

```
#!/usr/local/bin/perl


use strict;
use IO::File;

die "Usage: fireSlaves.pl #slaves " unless @ARGV > 0;

my($hostlist) = new IO::File("<thesis/tools/longhostlist") ||
  die "can't open longhostlist";

my(@hosts) = <$hostlist>;
@hosts = grep !/^#/, @hosts;
chomp @hosts;

die "not enough unique hosts!" unless $ARGV[0] <= @hosts ;

fisher_yates_shuffle(\@hosts);

map {
  print $hosts[$_] . "\n";
  system("ssh -x " . $hosts[$_]   " thesis/r/container \&");
} (2..$ARGV[0]);


# From perl FAQ
# fisher_yates_shuffle( \@array ) : generate a random
    permutation
# of @array in place
sub fisher_yates_shuffle {
    my $array = shift;
    my $i;
    for ($i = @$array; --$i; ) {
        my $j = int rand ($i+1);
        next if $i == $j;
```

```
            @$array[$i,$j] = @$array[$j,$i];
        }
}
```

A.2 Test Code

A.2.1 Lab Startup Scripts

The RIT Computer Science Department has 7 student-accessible laboratories. The ICL1 - ICL5, CSL, and GRAD scripts in the r directory start host containers on all machines in the eponymous lab. The scripts are:

```
#!/usr/local/bin/zsh
#now, start the other containers
icl1
echo Starting $i
ssh -x $i thesis/r/container &
echo $i started.
end

#!/usr/local/bin/zsh
#now, start the other containers
icl2
echo Starting $i
ssh -x $i thesis/r/container &
echo $i started.
end

#!/usr/local/bin/zsh
#now, start the other containers
icl3
echo Starting $i
ssh -x $i thesis/r/container &
echo $i started.
end

#!/usr/local/bin/zsh
#now, start the other containers
icl4
echo Starting $i
ssh -x $i  thesis/r/container &
echo $i started.
end

#!/usr/local/bin/zsh
#now, start the other containers
icl5
echo Starting $i
ssh -x $i thesis/r/container &
```

```
echo $i started.
end

#!/usr/local/bin/zsh
#now, start the other containers
csl
echo Starting $i
ssh -x $i thesis/r/container &
echo $i started.
end

#!/usr/local/bin/zsh
#now, start the other containers
gradlab
echo Starting $i
ssh -x $i thesis/r/container &
echo $i started.
end
```

A.2.2 DAM Access Collision Testing

The r/tortureMonitor script starts the JADE platform GUI on the local host, along with 20 sensory agents. Variations on this script were used to explore scalability issues pertaining to shared data structure access conflicts.

```
#!/usr/local/bin/zsh
java -Djava.security.policy= jini1_2_1/policy/policy.all\
  jade.Boot -container -port 20029 -host hilly.cs.rit.edu \
  rma:jade.tools.rma.rma \
  Bug01:stat.Bug Bug02:stat.Bug Bug03:stat.Bug \
  Bug04:stat.Bug Bug05:stat.Bug Bug06:stat.Bug \
  Bug07:stat.Bug Bug08:stat.Bug Bug09:stat.Bug \
  Bug10:stat.Bug Bug11:stat.Bug Bug12:stat.Bug \
  Bug13:stat.Bug Bug14:stat.Bug Bug15:stat.Bug \
  Bug16:stat.Bug Bug17:stat.Bug Bug18:stat.Bug \
  Bug19:stat.Bug
```

A.2.3 Duplication Testing

The tools/runduper script is a wrapper around the tools.DupePuller class, which determines whether the JavaSpace contains multiple entries with the given name.

```
#!/usr/local/bin/zsh
java -Djava.security.policy=jini1_2_1/policy/policy.all\
  tools.DupePuller $1 $2
```

```
package tools;

import net.jini.core.event.*;
import net.jini.space.JavaSpace;
import net.jini.core.lease.Lease;
import net.jini.core.transaction.server.TransactionManager;
import net.jini.core.entry.UnusableEntryException;
import net.jini.core.transaction.TransactionException;

import java.util.*;
import java.rmi.*;
import ns.dam.*;
import ns.infra.*;
import java.io.*;
import java.rmi.server.*;

public class DupePuller {

  static SpaceAccessor sa = new SpaceAccessor();
  static JavaSpace js = sa.getSpace();
  static DAMConnector dc = new DAMConnector(js);

  public static void main(String args[]) throws Exception
      {
    String name = args[0];
    System.out.print("dupechecking " + name);
    DAM d = dc.getDAM(name);
    if (d == null) {
      System.out.println(" not found");
      return;
    } else {
      DAE templ = d.matchableTemplate();
      System.out.println(" has template " + templ);
      System.out.println(" with tag " + templ.getTag());
      LinkedList l = new LinkedList();
      DAE temp = null;
      do {
        temp = (DAE)js.take(templ, null, JavaSpace.NO_WAIT);
        if (temp != null) {
          l.add(temp);
          System.out.println(temp);
        }
      } while (temp != null);

      System.out.println("restoring JavaSPace");
      while (l.size() > 0) {
        temp = (DAE)l.getFirst();
        js.write(temp, null, temp.getRemainingLeaseTime());
        l.removeFirst();
```

```
            System.out.println("restored " + temp);
          }
        }
      }
    }
```

A.2.4 InfoSpace Tab & Lock Performance Testing

The `tools/rundtstress` script is a wrapper around the `tools.dtstress` class. This class creates and fills test DAMs with randomly generated elements. In so doing, the dtstress tool performs 1000 locks and unlocks. This script was used to help ascertain the cause(s) of the write-locking scalability issues encountered in testbeds running CPs which created fierce contention for DAMs.

```
#!/usr/local/bin/zsh
java -Djava.security.policy=jini1_2_1/policy/policy.all
  tools.dtstress %1
```

```
package tools;

import net.jini.core.event.*;
import net.jini.space.JavaSpace;
import net.jini.core.lease.Lease;
import net.jini.core.transaction.server.TransactionManager;
import net.jini.core.entry.UnusableEntryException;
import net.jini.core.transaction.TransactionException;

import java.util.*;
import java.rmi.*;
import ns.dam.*;
import ns.infra.*;
import java.io.*;
import java.rmi.server.*;

public class dtstress {

  static JavaSpace js = (new SpaceAccessor()).getSpace();
  static DAMConnector dc = new DAMConnector(js);

  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer();
    List stuff = new LinkedList();

    System.out.println("GOBDAM master");
    DAM master = dc.GOBDAM(new DumbDAE(), "master",true);

    System.out.println("GOBDAM zero");
    DAM slave = dc.GOBDAM
```

```
        (new DumbDAE("zero", 1200000), "zero",false);

    System.out.println("GOBDAM one");
    DAM slave2 = dc.GOBDAM
        (new DumbDAE("one", 1200000), "one",false);

    System.out.println("GOBDAM two");
    DAM slave3 = dc.GOBDAM
        (new DumbDAE("two", 1200000), "two",false);

    System.out.println("GOBDAM three");
    DAM slave4 = dc.GOBDAM
        (new DumbDAE("three", 1200000), "three",false);

    DAM slaves[] = {slave, slave2, slave3, slave4};

    Random r = new Random((new Date()).getTime());

    try {
      System.out.println("built lists");
      master = dc.lock(master);
      System.out.println("master locked");
      master.add(slave,js);
      master.add(slave2,js);
      master.add(slave3,js);
      master.add(slave4,js);
      System.out.println("lists added");
      dc.unlock(master);
      System.out.println("master unlocked");
      int dest = 0;
      DAM temp = null;
      for (int i=0; i<100; i++) {
        for (int j=0; j<10; j++) {
          System.out.print(" ");
          dest = r.nextInt(slaves.length);
          temp = slaves[dest];
          temp = dc.lock(temp);
          temp.add(new DumbDAE
            ("", 1200000, dest + "::" + r.nextInt()),js);
          dc.unlock(temp);
        }
        System.out.print(" " + 10*i);
      }
    } catch (Exception e) { e.printStackTrace(); }
  }
}
```

A.3 Postprocessing Code

A.3.1 Host Statistics

The `showAllHostStats` script finds all host IAT statistics log files (results of Phase 2 statistics computation) in the current directory, sorts them, and passes the results into the `parseHostAvg.pl` program for data extraction (see Section 8.7.1). The resulting tables are plotted via Gnuplot.

```
#!/usr/bin/csh -f
#old naming scheme
#find ./ -name \*::DS-all-DS-IAT-processes\* \
#| sort | /thesis/tools/parseHostAvg.pl

#new naming scheme
find ./ -name \*::all-DS-all-DS-IAT\* \
  | sort | ~/thesis/tools/parseHostAvg.pl

gnuplot ~/thesis/tools/masterStatPlot.gnu
```

```
#!/usr/local/bin/perl

use strict;
use IO::File;

my($avg);
my($shtring);
my(@monotonic);
my(@temp);
my(@data);
my(@fn) = <>;

my(%dh);
my($file) ;
my(@avgs);
my(@alldata);
sub dateSort { $a->[2] cmp $b->[2]; }
sub avgSort { $a->[0] <=> $b->[0]; }

foreach (@fn) {
  $file = new IO::File("<$_");
  my(@data) = <$file>;
  my(@localarray);
  my($sum,$count);
  foreach (@data) {
    my($date, $tag, $rlt, $ct, $avg) =
    /date: '(.*2)' *tag: (\w*).*rlt. (\d+).*Ct: (\d+) Avg: (\
        d+(.\d*){0,1})/;

    @localarray = ($avg, $rlt, $date, $tag, $ct);
```

```
        next if ($dh{$tag});
        $dh{$tag} = 1;
        $sum += $avg;
        $count = $count + 1;
    }
    $avg = int ($sum/$count) unless ($count == 0);
    push @avgs, $avg;
    push @alldata, \@localarray;

}

my($i) = 1;

my($file) = new IO::File("> temporal");
map {
    $file->print( "$i @{$_}\n"); $i++;
} sort dateSort @alldata;
$file->close;

$i = 1;

@monotonic = sort avgSort @alldata;
$file = new IO::File("> monotonic");
map {$file->print("$i @{$_}\n"); $i++;} @monotonic;
$file->close;

$file = new IO::File("> monoStat");
map {$file->print("@{$_}->[0]\n");} @monotonic;
$file->close;

$file->open("cat monoStat | java tools.OSTB 0.9 0.05 |");
@data = <$file>;
$file->close;

print "%ile point: $data[0]\n";
map { print $data[$_];} (1..6);


$file = new IO::File(">90pctile.gnu") || die;
$file->print("pctile90_w_conf95=$data[0]");
$file->close;
```

A.3.2 tools/formMatrix.pl

The `tools/formMatrix.pl` script is intended to be executed from an experimenter's `archived-runs` directory as described in Section 8.5.4. It locates all run directories located beneath the current directory, extracts their QoS values, and compiles these values into a table suitable for importing into Matlab or another analysis tool. Note that if the experimenter wishes to merge data from

multiple archival directories, it is necessary to run the formMatrix tool in each of these directories, and manually combine the results.

```
#!/usr/local/bin/perl
$|=1;

use strict;

use IO::File;
use DirHandle;

my($ac); #agent count
my($cc); #container count

my($d) = new DirHandle(".");
my(@dirs) = map{$d->read} $d;
$d->close;
my($curFile);
my($sd); #subdirectory

foreach (@dirs) {
  my($curdir) = $_;
  next unless /^(\d*)c(\d*)a$/;
  $cc = $1;
  $ac = $2;
  $d = new DirHandle "./$curdir";

  my(@sdirs) = map{$d->read} $d;
  $d->close;

innerLoop: foreach $sd (@sdirs) {
    next innerLoop if $sd =~ /\./;
    $curFile = new IO::File("<$_/$sd/munchedStats") ||
          (warn "no stats for $_/$sd" && next innerLoop);
    my($line) = <$curFile>;
    $curFile->close;
    print log($cc)/log(2). "\t"
        log($ac)/log(2). "\t" .
        (split(" ",$line))[2]   "\n";
  }
}
```

A.3.3 tools/extractCutoffs & munchStats

The `tools/extractCutoffs` script executes performs postprocessing on archived run data. It computes QoS values for each archived run via the `tools/munchStats.sh` script, and formats them for analysis in Matlab or other tools with the `tools/formMatrix.pl` script.

```
#!/usr/local/bin/zsh
```

```
cd ~/archived-runs

foreach d (`find ./ -type d | grep -v "a$"`)
cd ~/archived-runs/${d}
echo ${d}
~/thesis/tools/munchStats > munchedStats
cd ~/archived-runs
end

echo stats computed

cd /archived-runs

perl ~/thesis/tools/formMatrix.pl

#!/usr/bin/csh -f

ls -lrt ./*DS-all-DS-IAT-processes* |\
   /thesis/tools/parseHostAvg.pl

#do not generate plots
#gnuplot ~/thesis/tools/masterStatPlot.gnu -
```

A.4 Analysis Code

A.4.1 tools.OSTB

The tools.OSTB class computes One Sided (Distribution Free) Tolerance Bounds given a population percentile, a confidence, and a series of data. The percentile and confidence are respectively the first and second command line arguments, while data is provided from the console. Note that the confidence is specified as the *error percentage*, so a user desiring a confidence of 95% will pass in a second command line argument of 0.05.

```
package tools;

import stat.*;
import java.util.*;
import java.io.*;
import cern.jet.stat.*;

public class OSTB {

//distribution-free confidence interval for a %ile
```

```
//Hahn & Meeker P83
//public static double[] epyp(int n, double p, double alpha)
    {
//public static double[] OSUCB(int n, double p, double alpha
    ) {
//public static int minSSizeForOSTB(double p, double alpha)
    {

  public static void main(String args[]) throws Exception
     {

    double pctile = Double.parseDouble(args[0]);
    double alpha  = Double.parseDouble(args[1]);
    Object data[]  = null;
    int cutoff = -1;
    int minSize = -1;;

    try {
      pctile = Double.parseDouble(args[0]);
      alpha  = Double.parseDouble(args[1]);
    } catch (ArrayIndexOutOfBoundsException e) {
      System.err.println
        ("usage: java tools.TestOSTB  <double>p <double>alpha"
          );
      return;
    }

    try {
      minSize = DistFree.minSSizeForOSTB(pctile,alpha);
      LinkedList dataL = new LinkedList();
      BufferedReader r = new BufferedReader
            (new InputStreamReader(System.in));

      String s = null;
      while ((s = r.readLine()) != null) {
        dataL.add(new Double(s));
      }

      data = dataL.toArray();

      Arrays.sort(data);

      double[] parms = DistFree.OSUCB(data.length,pctile,
          alpha);
      cutoff = (int)parms[1];

      System.out.println(data[cutoff-1]); //for the gnuplot
          script
      System.out.println("p: " +pctile);
      System.out.println("alpha: " +alpha);
```

```
        System.out.println("minsz: " +minSize);
        System.out.println("#data: " +data.length);
        System.out.println("conf : " +parms[0]);
        System.out.println("cutof: " + cutoff);
        for (int i=0; i<cutoff-1 && i < data.length; i++) {
          System.out.println(data[i]);
        }

        System.out.println("**********");
        System.out.println(data[cutoff-1]);
        System.out.println("**********");

        for (int i=cutoff; i<data.length; i++) {
          System.out.println(data[i]);
        }

        System.out.println();
      } catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("Insufficient data for p=" + pctile +
                " alpha: " + alpha + " confidence assessment.");
        System.err.println
          (data.length + " acquired; " +
           minSize + " required at minimum." );
      }
    }
}
```

A.4.2 globalPct.m

`globalPct.m` is a Matlab function which performs the basic analysis on the collection of all runs' QoS values. The function performs the following tasks:

1. Read in a matrix in the form output by `formMatrix.pl`
2. Display a 3-D scatter plot of the collected data
3. Compute the mean of each CP's QoS value set
4. Cubically interpolate a 3-D surface through the computed means
5. Label, format, and superimpose the scatter plot and interpolated surface.

```
%cd c:\pctileData\
pctiles = load ('percentiles', '-ascii');
xlin = linspace(min(pctiles(:,1)),max(pctiles(:,1)),50);
ylin = linspace(min(pctiles(:,2)),max(pctiles(:,2)),50);
pctiles(:,1) = 2.^pctiles(:,1);
pctiles(:,2) = 2.^pctiles(:,2);
pctiles(:,3) = 0.001.*pctiles(:,3);
```

```
figure
pctplot = scatter3(pctiles(:,1), pctiles(:,2), pctiles(:,3)
    , 15, 11*pctiles(:,1) + 17*pctiles(:,2), 'o', 'filled');
colormap(jet);
axis tight
grid on
hold on
[X,Y] = meshgrid(2.^xlin,2.^ylin);
[Xg,Yg,Zg] = griddata(pctiles(:,1),pctiles(:,2),pctiles(:,3)
    ,X,Y,'cubic');
scatterSurf = surf(X,Y,Zg);
set(scatterSurf, 'FaceColor', 'interp');
set(scatterSurf, 'EdgeColor', 'black');
set(scatterSurf, 'meshstyle', 'row');
set(scatterSurf, 'FaceColor', 'interp');
set(gca, 'XScale', 'log');
set(gca, 'YScale', 'log');
set(gca, 'ZScale', 'log');
hold off

xlabel 'Managed Hosts'
ylabel 'Sensory Agents'
zlabel '90%ile mean IAT upper DFIB (s)'
title 'Global percentile data by container/agent count'
```

A.4.3 straightRel.m

straightRel.m superimposes three planes on the current figure. These planes describe the Hypothesis 1 revised-form equation multiplied by three constant coefficients (45, 35, and 25).

```
hold on

tempS =surf(X,Y,45*(X./Y));
set(tempS,    'EdgeColor', [1 .6 .6])
set(tempS,    'FaceColor', [1 .6 .6]);
set(tempS, 'Marker', 'none');
set(tempS,'MarkerSize', 2);
set(tempS,'Clipping', 'on');

tempS =surf(X,Y,35*(X./Y));
set(tempS,    'EdgeColor', [.6 1 .6])
set(tempS,    'FaceColor', [.6 1 .6]);
set(tempS, 'Marker', 'none');
set(tempS,'MarkerSize', 2);
set(tempS,'Clipping', 'on')

tempS =surf(X,Y,25*(X./Y));
set(tempS,    'EdgeColor', [.6 .6 1])
```

```
set(tempS,    'FaceColor', [.6 .6 1]);
set(tempS, 'Marker', 'none');
set(tempS,'MarkerSize', 2);
set(tempS,'Clipping', 'on');;
hold off
```

A.4.4 MVregress.m

MVregress.m performs a linear, least-squares multivariate regression on the collection of means of CP QoS value sets. At present, the means are computed by manually specifying the starting and ending indices of the CP datasets within the pctiles table. Once the means are computed, the regression is performed and the resultant coefficients are used to generate a regression surface, which is superimposed on the current figure.

```
hold on
%This must be updated if new data are added.
MANAVGS = [
pctiles(1,1:2) mean(pctiles(1:15,3));
pctiles(16,1:2) mean(pctiles(16:29,3));
pctiles(30,1:2) mean(pctiles(30:42,3));
pctiles(43,1:2) mean(pctiles(43:56,3));
pctiles(57,1:2) mean(pctiles(57:61,3));
pctiles(62,1:2) mean(pctiles(62:69,3));
pctiles(70,1:2) mean(pctiles(70:79,3));
pctiles(80,1:2) mean(pctiles(80:80,3));
pctiles(81,1:2) mean(pctiles(81:88,3));
pctiles(89,1:2) mean(pctiles(89:98,3));
pctiles(99,1:2) mean(pctiles(99:102,3));
pctiles(103,1:2) mean(pctiles(103:107,3));
pctiles(108,1:2) mean(pctiles(108:112,3));
pctiles(113,1:2) mean(pctiles(113:114,3))];

fooX = [ones(size(MANAVGS(:,1))) MANAVGS(:,1) MANAVGS(:,2)];
fooA = fooX\MANAVGS(:,3)
fooY = fooX*fooA;
fooMaxErr = max(abs(fooY - MANAVGS(:,3)))
tempS = surf(X,Y,fooA(1) + fooA(2)*X + fooA(3)*Y);
set(tempS,    'EdgeColor', [.6 .6 .6])
set(tempS,    'FaceColor', 'none');
set(tempS, 'Marker', 'none');
set(tempS,'MarkerSize', 2);
set(tempS,'Clipping', 'on');

hold off
```

A.5 Utility Code

A.5.1 runjsi / jsi

The `tools.jsi` program produces a textual representation of a specifed DAM, recursively enumerating all nested DAMs until either no further DAMs are found or the optionally specified maximum depth is reached. `tools/runjsi` is a wrapper around the `tools.jsi` class. Both class and script accepts as command line arguments the target DAE name (a String) optionally followed by a nonnegative integer indicating the number of levels to recurse through.

```
#!/usr/local/bin/zsh
java -Djava.security.policy=jini1_2_1/policy/policy all \
  tools.jsi $1 $2
```

```
package tools;

import net.jini.core.event.*;
import net.jini.space.JavaSpace;
import net.jini.core.lease.Lease;
import net.jini.core.transaction.server.TransactionManager;
import net.jini.core.entry.UnusableEntryException;
import net.jini.core.transaction.TransactionException;

import java.util.*;
import java.rmi.*;
import ns.dam.*;
import ns.infra.*;
import java.io.*;
import java.rmi.server.*;

public class jsi {

  static SpaceAccessor sa = new SpaceAccessor();
  static JavaSpace js = sa.getSpace();
  static DAMConnector dc = new DAMConnector(js);

  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer();
    int maxdepth = Integer.MAX_VALUE;
    try {
      maxdepth = Integer.parseInt(args[1]);
      System.out.println("max depth:  " + maxdepth);
    }
    catch (NullPointerException e) {}
    catch (ArrayIndexOutOfBoundsException e) {}
     catch (Exception f) {f.printStackTrace();}

    tree(args[0],0,sb,maxdepth);
```

```
    System.out.println(sb.toString());
  }

  public static void tree
    (String name, int level, StringBuffer sb, int maxdepth) {
    System.out.print(name);
    DAM d = dc.getDAM(name);
    if (d == null) {
      System.out.println(" not found");
      return;
    } else {
      System.out.println(" has " + d.getSize() + " elements");
    }
    for (int i=0; i<level; i++) sb.append("\t");

    sb.append(d + " : " +
      Long.toString(d.getRemainingLeaseTime()) + "\n");
    if (level < maxdepth) {
      Iterator api = d.iterator(js);
      SortedSet ss = new TreeSet();
      DAE temp = null;

      while (api.hasNext()) {
        temp = (DAE)api.next();
        if (temp != null) ss.add(temp);
      }

      Iterator ssi = ss.iterator();

      for (int q=0; ssi.hasNext(); q++) {
        DAE dae = (DAE)ssi.next();
          if (dae instanceof DAM) {
            System.out.print(level + ":" +q + " ");
            tree(((DAM)dae).getName(),level+1,sb, maxdepth);
          } else {
            for (int i=0; i<level+1; i++) sb.append("\t");
            sb.append(dae + " : " +
            (dae == null? "expired" :
             Long.toString(dae.getRemainingLeaseTime())) + "\n
                 ");
          }
        }
      System.out.println();
    }
  }
}
```

A.5.2 Cleanup

Occasionally, one or more host containers will fail to terminate when requested to do so. Because Net.Sense assumes a bijection between managed hosts and JADE containers, all containers from a given run must be terminated before a new run can be initiated. This script automates the process of cleaning up after a Net.Sense run by serially connecting to each (potentially) managed host and killing any of the user's processes found thereon.

```
#!/usr/local/bin/zsh

echo CSL start
csl
ssh -x $i kill -9 -1 &
end

sleep 10

echo gradlab start

gradlab
ssh -x $i kill -9 -1 &
end

sleep 10

echo icl5 start

icl5
ssh -x $i kill -9 -1 &
end

sleep 10

echo icl4 start
icl4
ssh -x $i kill -9 -1 &
end

sleep 10

echo icl3 start
icl3
ssh -x $i kill -9 -1 &
end

sleep 10

echo icl2 start
icl2
```

```
ssh -x $i kill -9 -1 &
end
sleep 10

echo icl1 start
icl1
ssh -x $i kill -9 -1 &
end
```

A.5.3 tools/runClean

The tools/runClean script cleans up unwanted process data log files while Net.Sense is running. While these files are of interest in some cases, these log files can consume disk space at significant rate. Since the actual process data are not particularly useful when studying Net.Sense system performance, the author elected to remove these files as they were created during operation.

```
#!/usr/local/bin/zsh
while (true) do
ls ~/run/Bug*Solaris*
/usr/bin/rm -f ~/run/Bug*Solaris*
date
sleep 60
done
```

Bibliography

[1] Brookings Institute's Ascape home page. `http://www.brook.edu/dybdocroot/es/dynamics/models/ascape/`.

[2] Expert systems case studies: Mycin. `http://www.computing.surrey.ac.uk/research/ai/PROFILE/mycin.html`.

[3] FIPA home page. `http://www.fipa.org`.

[4] J. S. Aikins, J. C. Kunz, E. H. Shortliffe, and R. J. Fallat. PUFF: An expert system for interpretation of pulmonary function data. *Computers and Biomedical Research*, 16:199–208, 1983.

[5] Jarmo T. Alander. An indexed bibliography of genetic algorithms and neural networks. `ftp://ftp.uwasa.fi/cs/report94-1/gaTSPbib.ps.Z`.

[6] Jarmo T. Alander. An indexed bibliography of genetic algorithms and the traveling salesman problem. `ftp://ftp.uwasa.fi/cs/report94-1/gaNNbib.ps.Z`.

[7] Debra Anderson, Thane Frivold, Ann Tamaru, and Alfonso Valdes. Next-generation Intrusion Detection Expert System (NIDES), software users manual, beta-update release. Technical Report SRI–CSL–95–07, Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025-3493, May 1994.

[8] B. Arnold, A. van Deursen, and M. Res. An algebraic specification of a language for describing financial products. In M. Wirsing, editor, *ICSE-17 Workshop on Formal Methods Application in Software Engineering*, pages 6–13. IEEE, April 1995.

[9] David Eby Averill. The optimization of flywheels using an injection island genetic algorithm. In P. Bentley, editor, *Evolutionary Design by Computers*, pages 167–190. Morgan Kaufmann, 1999.

[10] Pilar Barrufet, Josep Puyol-Gruart, and Carles Sierra. Terap-IA, a knowledge-based system for pneumonia treatment. In *Proceedings of the International ICSC Symposium on Engineering of Intelligent Systems (EIS'98)*, volume 1, pages 176–182, 1998.

[11] Michael Beetz. Structured reactive controllers: controlling robots that perform everyday activity. In *Proceedings of the third annual conference on Autonomous Agents*, pages 228–235. ACM Press, 1999.

[12] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. JADE a FIPA-compliant agent framework. In *Proceedings of PAAM'99*, pages 97–108, April 1999.

[13] J. G. Bellingham and T. R. Consi. State configured layered control. In *1st Workshop on Mobile Robots for Subsea Environments*, pages 75 – 80, October 1990.

[14] Forrest H Bennett III, John R. Koza, David Andre, and Martin A. Keane. Evolution of a 60 decibel op amp using genetic programming. In Tetsuya Higuchi, Masaya Iwata, and Weixin Liu, editors, *ICES*, volume 1259 of *Lecture Notes in Computer Science*, pages 312–326. Springer, 1996.

[15] G. Boella, R. Damiano, and L. Lesmo. Cooperating to the group's utility. In *Proceedings of the 6th International Workshop on Agent Theories, Architectures, and Languages (ATAL-99)*, pages 319 – 333. Springer-Verlag, 2000.

[16] F. Bond, T. Yamazaki, R. Sulong, and K. Ogura. Design and construction of a machine-tractable Japanese-Malay lexicon. In *7th Annual Meeting of the Association for Natural Language Processing*, pages 62–65, 2001.

[17] Gary Boone. Concept features in re:agent, an intelligent email agent. In *Autonomous Agents*, pages 141–148, Minneapolis, MN, 1998. Association of Computing Machinery, ACM.

[18] A. Broder and A. Karlin. Bounds on the cover time. In *Journal of Theoretical Probability*, volume 2, pages 101 – 120, 130, Lytton Ave., Palo Alto, CA 94301, USA, 1989.

[19] Andrei Z. Broder, Anna R. Karlin, Prabhakar Raghavan, and Eli Upfal. Trading space for time in undirected s-t connectivity. In *ACM Symposium on Theory of Computing*, pages 543–549, 1989.

[20] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.

[21] R. A. Brooks. A hardware retargetable distributed layered architecture for mobile robot control. *Proc. of IEEE International Conference on Robotics and Automation, NC*, pages pp. 106–110, 1987.

[22] Rodney A. Brooks. Elephants don't play chess. *Robotics and Autonomous Systems*, 6(1&2):3 – 15, June 1990.

[23] Wolfram Burgard, Armin B. Cremers, Dieter Fox, Dirk Hahnel, Gerhard Lakemeyer, Dirk Schulz, Walter Steiner, and Sebastian Thrun. The interactive museum tour-guide robot. In *AAAI/IAAI*, pages 11–18, 1998.

[24] Paolo Busetta and Kotagiri Ramamohanarao. An architecture for mobile BDI agents. In *Proceedings of the 1998 ACM symposium on Applied Computing*, pages 445–452. ACM Press, 1998.

[25] Greg Butler, Andrea Gantchev, and Peter Grogono. Reusable strategies for software agents via the subsumption architecture. In *Proceedings of the 6th Asia Pacific Software Engineering Conference*, pages 326 – 334. IEEE, 1999.

[26] FIPA Technical Committee C. FIPA ACL message structure specification. Technical Report 61, FIPA, August 2001.

[27] Neil A. Campbell and Jane B. Reece. *Biology*. Benjamin/Cummings, 6 edition, 2002. Chapters 22–24.

[28] Norman Carver and Victor Lesser. The evolution of blackboard control architectures. Technical Report UM-CS-1992-071, University of Massachusetts, 1992.

[29] Liren Chen and Katia Sycara. WebMate: A personal agent for browsing and searching. In Katia P. Sycara and Michael Wooldridge, editors, *Proceedings of the 2nd International Conference on Autonomous Agents (Agents'98)*, pages 132–139, New York, 1998. ACM Press.

[30] D. Cliff, P. Husbands, and I. Harvey. Evolving visually guided robots. In J-A Meyer, H. Roitblat, and S. Wilson, editors, *From Animals to Animats: Proceedings of the Second International Conference on Simulation of Adaptive Behaviour (SAB92)*, pages 374–383. Springer-Verlag, 1993.

[31] A. Colorni, M. Dorigo, and V. Maniezzo. Distributed optimization by ant colonies. In *Proceedings of ECAL91 European Conference on Artificial Life*, pages 134–142. Elsevier, 1991.

[32] Galois Connections. Cryptol home page. `http://www.cryptol.net`.

[33] R. Scott Cost, Ian Soboroff, Jeegar Lakhani, Timothy W. Finin, Ethan L. Miller, and Charles Nichols. TKQML: A scripting tool for building agents. In *Agent Theories, Architectures, and Languages*, pages 339 – 343, 1997.

[34] I. D. Craig. From blackboards to agents. In *Online Proceedings of the VIM Project Spring Workshop on Collaboration Between Human and Artificial Societies*, 1998. `http://vim.ecs.soton.ac.uk/`.

[35] Kenneth Dawson-Howe. Autonomous probing robots for the detection of abandoned landmines. In *Proceedings of the 5th Symposium on Intelligent Robotics Systems (SIRS-97)*, pages 51–58, July 1997.

[36] Nicolette de Bruijn, Peter Lucas, Karin Schurink, and Andy Hoepelman. Improving antibiotic therapy of ventilator associated pneumonia using a probabilistic approach. Technical Report UU-CS-1999-06, Universiteit Utrecht, 1999.

[37] Tadeusz P. Dobrowiecki, Gyrgy Strausz, and Tams Mszros. Knowledge fusion for financial advisory applications. In *The 7th Biennial Conference on Electronics and Microsystem Technology, Baltic Electronics Conference, BEC 2000*, October 2000.

[38] C. Henry. Edwards and David E. Penney. *Differential Equations and Boundary Value Problems.* Prentice Hall, 2nd edition, 2002.

[39] L. D. Erman, P. E. London, and S. F. Fickas. The design and an example use of HEARSAY-III. In *Proceedings of the National Conference on Artificial Intelligence*, pages 409–415. MIT Press, 1983.

[40] Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser, and D. Raj Reddy. The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty. *ACM Computing Surveys (CSUR)*, 12(2):213 – 253, 1980.

[41] S. Falasconi and M. Stefanelli. A library of medical ontologies. In N. J. I. Mars, editor, *Proceedings of the ECAI94 Workshop Comparison of Implemented Ontologies*, pages 81–91, 1994.

[42] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an Agent Communication Language. In N. Adam, B. Bhargava, and Y. Yesha, editors, *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM'94)*, pages 456–463, Gaithersburg, MD, USA, 1994. ACM Press.

[43] Tim Finin. UMBC KQML Web. `http://www.cs.umbc.edu/kqml/`.

[44] Richard E. Flathman and David Johnston, editors. *Leviathan: authoritative text, backgrounds, interpretations.* W. W. Norton & Company, 1997.

[45] Agentcities Task Force. Agentcities web. `http://www.agentcities.net/index.jsp`.

[46] Jerry Fowler, Brad Perry, Marian H. Nodine, and Bruce Bargmeyer. Agent-based semantic interoperability in InfoSleuth. *SIGMOD Record*, 28(1):60–67, 1999.

[47] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice.* Addison-Wesley, 1999.

[48] R. J. Gallimore, N. R. Jennings, H. S. Lamba, C. L. Mason, and B. J. Orenstein. 3D scientific data interpretation using cooperating agents. In *Proceedings of the 3rd international conference on the practical applications of agents and multi-agent systems (PAAM-98)*, pages 47 – 65, London, UK, 1998.

[49] F. Gandon. Engineering an ontology for a multi-agents corporate memory system. In *Proceedings of ISMICK'01*, pages 209–228, 2001.

[50] A. Gangemi, G. Steve, and F. Giacomelli. ONIONS: An ontological methodology for taxonomic knowledge integration. In *ECAI-96 Workshop on Ontological Engineering*, pages 5 – 16, August 1996.

[51] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Longman, Inc, 1989.

[52] Anthony J. F. Griffiths, Jeffery H. Miller, David T. Suzuki, Richard C. Lewontin, and William M. Gelbart. *An Introduction to Genetic Analysis.* W. H. Freeman and Company, 7th edition, 2000. Chapter 24.

[53] Web-Ontology (WebOnt) Working Group. Working group home page. `http://www.w3.org/2001/sw/WebOnt/`.

[54] T. R. Gruber. Towards principles for the design of ontologies used for knowledge sharing. In N. Guarino and R. Poli, editors, *Formal Ontology in Conceptual Analysis and Knowledge Representation*, Deventer, The Netherlands, 1993. Kluwer Academic Publishers.

[55] Gerald J. Hahn and William Q. Meeker. *Statistical Intervals: A Guide for Practitioners.* John Wiley & Sons, 1991.

[56] A. L. G. Hayzelden and J. Bigham. Heterogenous multi-agent architecture for ATM virtual path network resource configuration. In S. Albayrak and F. J. Garijo, editors, *Intelligent Agents for Telecommunication Applications — Proceedings of the Second International Workshop on Intelligent Agents for Telecommunication*, volume 1437, pages 45 – 59. Springer-Verlag: Heidelberg, Germany, 1998.

[57] C. Heinze, S. Goss, I. Lloyd, and A. Pearce. Plan recognition in military simulation: Incorporating machine learning with intelligent agents. In *Proceedings of IJCAI-99 Workshop on Team Behaviour and Plan Recognition*, pages 53–64, 1999.

[58] D. E. Hirsch. An expert system for diagnosing gait in cerebral palsy patients. Technical Report TR-388, MIT LCS, 1987.

[59] J. Huang, N. R. Jennings, and J. Fox. An agent architecture for distributed medical care. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents: Theories, Architectures, and Languages (LNAI Volume 890)*, pages 219–232. Springer-Verlag: Heidelberg, Germany, 1995.

[60] Sam Joseph and T Kawamura. Why autonomy makes the agent. In J. Liu, N. Zhong, Y.Y. Tang, and P. Wang, editors, *Agent Engineering.* World Scientific Publishing, 2001.

[61] M. Keith. A blackboard system for automatic transcription of simple polyphonic music. Perceptual Computing Technical Report 385, MIT Media Lab, July 1996.

[62] James Kennedy, Russell C Eberhart, and Yuhui Shi. *Swarm Intelligence.* Morgan Kaufmann Publishers, 2001.

[63] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. RoboCup: The robot world cup initiative. In W. Lewis Johnson and Barbara Hayes-Roth, editors, *Proceedings of the First International Conference on Autonomous Agents (Agents'97)*, pages 340–347, New York, 1997. ACM Press.

[64] John R. Koza, Forrest H. Bennett III, David Andre, and Martin A. Keane. Automated WYWIWYG design of both the topology and component values of electrical circuits using genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 123–131, Stanford University, CA, USA, 1996. MIT Press.

[65] C. Ronald Kube and Hong Zhang. Collective robotic intelligence. In *International Conference on Simulation of Adaptive Behaviour*, pages 460–468, 1992.

[66] P. N. Kugler and M. T. Turvey. Self-organization, flow fields, and information. *Human Movement Science*, 7:97 – 129, 1988.

[67] Y. Labrou, T. Finin, and Y. Peng. The current landscape of agent communication languages. *Intelligent Systems*, 14(2):45–52, March/April 1999.

[68] David A. Ladd and J. Christopher Ramming. Two application languages in software production. In *USENIX 1994 Very High Level Languages Symposium*, pages 169–177, 1994.

[69] P. Larraaga, C.M.H. Kuijpers, M. Poza, and R.H. Murga. Optimal decomposition of Bayesian networks by genetic algorithms. Technical Report EHU-KZAA-IK-3-94, University of the Basque Country, 1994.

[70] Tim Berners Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.

[71] Ramiro Liscano, Reda E. Fayek, Allan Manz, Elizabeth R. Stuck, and Jean-Yves Tigli. Using a blackboard to integrate multiple activities and achieve strategic reasoning for mobile-robot navigation. *IEEE Expert*, 10:24–36, 1995.

[72] George F. Luger and William A Stubblefield. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving.* Addison-Wesley Longman, Inc, 1998.

[73] Pattie Maes. Talk: Interacting with virtual pets and other software agents. `http://www.mediamatic.nl/doors/Doors2/Maes/Maes-Doors2-E.html`, 1994. Delivered at the Second *Doors of Perception* Conference in Amsterdam, The Netherlands.

[74] K. Mahesh. Ontology development for machine translation: Ideology and methodology. Technical Report MCCS-96-292, NM State University Computing Research Laboratory, 1996.

[75] K. Mahesh and S. Nirenburg. Semantic classification for practical natural language processing. In Ray Schwartz, editor, *Proceedings of the 6th ASIS SIG/CR Classification Research Workshop: An Interdisciplinary Meeting*, pages 79 – 94. Learned Information, 1998.

[76] Maja J Mataric. Integration of representation into goal-driven behavior-based robots. *IEEE Transactions on Robotics and Automation*, 8(3):304–312, June 1992.

[77] Jan Murray, Frieder Stolzenburg, Oliver Obst, and Björn Bremer. RoboLog Koblenz: Complex agent scripts implemented in logic. In Stefan Sablatnög and Stefan Enderle, editors, *Proceedings of the Workshop RoboCup during KI'99 in Bonn*, pages 12–25, 1999.

[78] Sanguk Noh and Piotr Gmytrasiewicz. Implementation and evaluation of rational communicative behavior in coordinated defense. In Oren Etzioni, Jörg P. Müller, and Jeffrey M. Bradshaw, editors, *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, pages 123–130, Seattle, WA, USA, 1999. ACM Press.

[79] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. In *Advances in Computers*, volume 46, pages 329 – 400. Academic Press, 1998.

[80] Barney Pell, Douglas E. Bernard, Steve A. Chien, Erann Gat, Nicola Muscettola, P. Pandurang Nayak, Michael D. Wagner, and Brian C. Williams. An autonomous spacecraft agent prototype. In W. Lewis Johnson and Barbara Hayes-Roth, editors, *Proceedings of the First International Conference on Autonomous Agents (Agents'97)*, pages 253–261, New York, 1997. ACM Press.

[81] Domenico M. Pisanelli, Aldo Gangemi, and Geri Steve. A medical ontology library that integrates the UMLS Metathesaurus (tm). *Lecture Notes in Computer Science*, 1620:239–248, 1999.

[82] Phillip A. Porras and Peter G. Neumann. EMERALD: Event Monitoring Enabling Responses To Anomalous Live Disturbances. In *Proceedings of the 20th NISSC*. NIST, October 1997. `http://csrc.nist.gov/nissc/1997/proceedings`.

[83] S. Quaglini, R. Bellazzi, M. Stefanelli, and F. Locatelli. Sharing and reusing therapeutic knowledge for managing leukemic children. In R. Engelbretch, S. Andreassen, and J. Wyatt, editors, *Artificial Intelligence in Medicine, Proc. of AIME 93 - 4th Conference on Artificial Intelligence in Medicine Europe*, pages 319 – 330. IOS Press, 1993.

[84] A. S. Rao and M. P. Georgeff. BDI-agents: from theory to practice. In *Proceedings of the First International Conference on Multiagent Systems*, pages 312 – 319, San Francisco, 1995.

[85] Laurent Reveillere, Fabrice Merillon, Charles Consel, Renaud Marlet, and Gilles Muller. A DSL approach to improve productivity and safety in device drivers development. In *Automated Software Engineering*, pages 101–110, 2000.

[86] J. Romein, H. Bal, and D. Grune. An application domain specific language for describing board games. In *Parallel and Distributed Processing Techniques and Applications*, volume I, pages 305–314, 1997.

[87] M. Schneider-Fontan and M. Mataric. A study of territoriality: The role of critical mass in adaptive task division. In *Animals to Animats IV, Fourth International Conference on Simulation of Adaptive Behavior (SAB-96),*, pages 553–561. MIT Press/Bradford Books, 1996.

[88] E. H. Shortliffe, F. S. Rhame, S. G. Axline, S. N. Cohen, B. G. Buchanan, R. Davis, A. C. Scott, R. Chavez-Pardo, and W. J. van Melle. MYCIN: A computer program providing antimicrobial therapy recommendations. Technical Report 107a, Stanford University, 1975. Originally published in *Clinical Research* 23:107a, 1975 (Not available).

[89] S. Skiena. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Addison-Wesley, 1990.

[90] Yannis Smaragdakis and Don Batory. DiSTiL: A transformation library for data structures. In *Proceedings of the Conference on Domain-Specific Languages*, pages 257–270. USENIX, October 1997.

[91] Grant A. E. Soremekun. Genetic algorithms for composite laminate design and optimization. Master's thesis, Virginia Polytechnic Institute, Blacksburgh, Virginia, May 1997.

[92] John Sowa. Ontology home page. `http://users.bestweb.net/~sowa/ontology/`.

[93] The Linda Group. Linda home page. `http://www.cs.yale.edu/Linda/linda.html`.

[94] Scott Thibault, Renaud Marlet, and Charles Consel. A domain-specific language for video device drivers: from design to implementation. In *1st USENIX conference on Domain-Specific Languages (DSL '97)*, pages 11–26, October 1997.

[95] M. Uschold, M. King, S. Moralee, and Y. Zorgios. The enterprise ontology. *The Knowledge Engineering Review*, 13:31 – 89, 1998.

[96] Patrick van Bommel. A randomized schema mutator for evolutionary database optimization. *Australian Computer Journal*, 25(2):61–69, 1993.

[97] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[98] Gertjan van Heijst, Sabina Falasconi, Ameen Abu-Hanna, Guus Schreiber, and Mario Stefanelli. A case study in ontology library construction. *Artificial Intelligence in Medicine*, 7(3):227–255, 1995.

[99] Richard T. Vaughan, Kasper Støy, Gaurav S. Sukhatme, and Maja J. Mataric. Whistling in the dark: Cooperative trail following in uncertain localization space. In Carles Sierra, Maria Gini, and Jeffrey S. Rosenschein, editors, *Proceedings of the Fourth International Conference on Autonomous Agents*, pages 187–194, Barcelona, Catalonia, Spain, 2000. ACM Press.

[100] B. Venkatesh and S. Goyal. Learning from neighbors. *Review of Economic Studies*, 65:595–621, 1998.

[101] F. von Martial. *Lecture Notes in Computer Science: Coordinating Plans of Autonomous Agents*. Springer-Verlag, 1992.

[102] I. Wagner and A. Bruckstein. Cooperative cleaners: a study in ant robotics. CIS Report 9512, Technion, IIT, Haifa, Israel, June 1995.

[103] I. Wagner, M. Lindenbaum, and A. Bruckstein. Cooperative covering by ant-robots using evaporating traces. Technical Report CIS-9610, Center for Intelligent Systems, Technion, April 1996. to appear in IEEE Trans. Rob. Aut.

[104] Israel A. Wagner and Alfred M. Bruckstein. Hamiltonian(t) - an ant-inspired heuristic for recognizing hamiltonian graphs. In Peter J. Angeline, Zbyszek Michalewicz, Marc Schoenauer, Xin Yao, and Ali Zalzala, editors, *Proceedings of the Congress on Evolutionary Computation*, volume 2, pages 1465–1469, Mayflower Hotel, Washington D.C., USA, 1999. IEEE Press.

[105] Israel A. Wagner, Michael Lindenbaum, and Alfred M. Bruckstein. Efficiently searching a graph by a smell-oriented vertex process. *Annals of Mathematics and Artificial Intelligence*, 24(1-4):211–223, 1998.

[106] Israel A. Wagner, Michael Lindenbaum, and Alfred M. Bruckstein. ANTS: Agents, networks, trees, and subgraphs. *Future Generation Computer Systems*, 16(8):915–926, June 2000.

[107] Robin J. Wilson and John J. Watkins. *Graphs: An Introductory Approach*. John Wiley & Sons, 1990.

[108] Winkler and Zuckerman. Multiple cover time. *RSA: Random Structures & Algorithms*, 9, 1996.

[109] Michael Wooldridge and Nicholas R. Jennings. Agent theories, architectures, and languages: A survey. In Michael Wooldridge and Nicholas R. Jennings, editors, *Intelligent Agents: Lecture Notes in AI*, volume 890, pages 1–39. Springer-Verlag, 1994.

[110] J. Yang. *Co-ordination Based Structured Parallel Programming.* PhD thesis, Imperial College of Science, Technology and Medicine, University of London, 1998.

[111] Jochen Zeil, Almut Kelber, and Rdiger Voss. Structure and function of learning flights in bees and wasps. *The Journal of Experimental Biology*, 199:245–252, 1996.