SodaBot: A Software Agent Construction System*

Michael H. Coen
MIT AI Lab
545 Technology Square
Cambridge, MA 02139
mhcoen@ai.mit.edu

Abstract

SodaBot is an environment for creating and using distributed software agents. The SodaBot system is composed of a virtual machine called the Basic Software Agent (BSA) and a language for programming it called SodaBotL. People write programs in SodaBotL which then run within a BSA. SodaBotL is a simple, very high level programming language. It is interpreted, highly-portable, and multi-threaded. Many of the familiar DAI inter-agent social-behaviors are implemented, including ask, request, command, inform, refuse, and reply.

The SodaBot system was designed with two purposes in mind: (1) to direct a flow of online information among its many sources and consumers and (2) to act as computational glue for interconnecting the denizens of the computational world: people, software, and hardware. We are interested both in exploring tasks traditionally done by software agents like *taskbots* (e.g., meeting scheduling) and in trying to generate new types of applications for the software agent community. We are also interested in exploring and implementing distributed (or agent-based) versions of classic AI paradigms, in particular parallel planning among interacting agents.

This paper presents some details of the SodaBot system along with some current applications.

Content Area: software agents

This is a draft. Please do not cite or distribute.

*This material is based upon work supported by the Advanced Research Projects Agency of the Department of Defense under contract number F30602–94–C–0204, monitored through Rome Laboratory and Griffiss Air Force Base, and conducted under the supervision of Professor Lynn Andrea Stein. Professor Stein's research is also supported by the Mitsubishi Electric Research Laboratories; by Sun Microsystems, Inc.; and by the National Science Foundation under National Science Foundation Young Investigator Award Grant No. IRI–9357761.

1 Introduction

SodaBot is an environment for creating and using distributed software agents.[1] The SodaBot system is composed of a virtual machine called the Basic Software Agent (BSA) and a language for programming it called SodaBotL.[2] People write programs in SodaBotL which then run within a BSA.

Typically, we associate each BSA with a particular person, object, or function, and we think of that entity as the BSA's owner. For example, I have a personal BSA which runs in the background on my home workstation, and the MIT AI Lab's Intelligent Room has a BSA which controls the room's many video displays.

SodaBotL is a simple, very high level programming language. It is interpreted, highly-portable, and multi-threaded. Many of the familiar DAI inter-agent social-behaviors are implemented, including ask, request, command, inform, refuse, and reply. SodaBotL conforms to our belief that essential to writing behaviorally complex, distributed agents is the ability to abstractly describe what the agents are doing in very high level terms. This simplifies both writing and reasoning about the agent's behavior.

The SodaBot system was designed with two purposes in mind: (1) to direct a flow of online information among its many sources and consumers and (2) to act as computational glue for interconnecting the denizens of the computational world: people, software, and hardware. We are interested both in exploring tasks traditionally done by software agents like *taskbots* ([Kautz *et al.*, 1994]) (e.g., meeting scheduling) and in trying to generate new types of applications for the software agent community. We are also interested in exploring and implementing distributed (or agent-based) versions of classic AI paradigms, in particular parallel planning and information retrieval among interacting agents using the techniques of [Etzioni *et al.*, 1992].

This paper presents details of the SodaBot system along with some current applications.

2 Related Work

Many systems are currently being released which support what is known as mobile code. For example, Telescript ([White, 1994]) is an environment for writing highly portable and secure code. It has an object-oriented programming language which provides a *go* operator that can move a running program and its state to a new host-machine during execution.

Java ([Sun, 1995]) is a multi-threaded highly-portable variant of C++ which

[1] See [AAAI, 1994] and [CACM, 1994] for surveys of current work in software agents.

[2] This is pronounced "Soda Bottle."

has gained popularity via its incorporation in several Web browsers. Although it is not per se a mobile code system, it is not difficult to build one using Java as a substrate. Among its many features, Java allows programmers to treat networked resources, e.g., TCP/IP connections, as if they were simply objects on a local file system.

Other transportable code systems include Obliq ([Cardelli, 1994]), which is notable for supporting networked continuations, and AgentTCL ([Gray, 1995]), which is roughly a TCL equivalent of Telescript.

SodaBot differs from these systems by providing much higher-level and wider-ranging facilities to agent programmers. For example, in SodaBot it is easy for a programmer to write inter-agent communication without knowing any details of what connections are made to whom or how to interface with the network. To communicate with a BSA, you simply have to know its name. Suppose we wanted to send a notification to our colleague David whose BSA is named *david@foo.bar.edu*.[3] In SodaBotL, we could write this without knowing anything other than his BSA's name:

```
$place = Agent Request assistant::notify($message)
           -user david@foo.bar.edu;
```

This would locate and contact David's *personal assistant* agent and ask it to notify him of the object contained in the variable `$message`. How the agent notifies him is left to its discretion. It might read the message out loud, fax it, etc., depending on the agent's configuration and knowledge of David's current status.

We could also write this somewhat more intrusively and in the spirit of Telescript by:

```
Agent hop david@foo.bar.edu;
Web display $message;
Agent hop back;
```

The SodaBotL *Agent hop* command is quite similar to the Telescript *go* command. Here, the running SodaBotL program (and its local state) would actually move to David's BSA, display the message in his Web browser, and then return to its original BSA. Of course, this would lose the benefit provided by asking his personal assistant to notify him of the message, because it assumes that displaying the message in his web browser is the proper thing to do. (Security considerations will be discussed below.)

[3] A BSA name does not necessarily correspond to an e-mail address. See section 3.1.

3 The SodaBot System

The primary component of the SodaBot system is the basic software agent (BSA). It is a virtual machine which runs application agents written by SodaBot users. Each of these users generally has a personal BSA which runs in the background of their home workstation and can follow them around if they move among different machines. Objects of interest in the computational world such as databases, speech recognition systems, and robots can also have personal BSAs that allow other BSAs and people to access them.

Although multiple BSAs can run together on a single machine, only one BSA is required per person/object because an individual BSA can run multiple SodaBotL programs simultaneously. The BSA is multi-threaded and implements a time sharing and event-driven scheduler to divide its attention among its various agents (which can themselves be multi-threaded.)

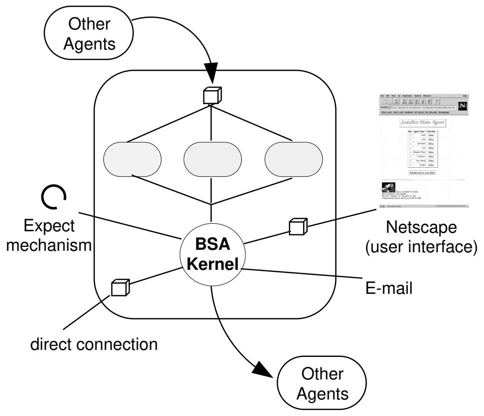

Figure 1: The BSA architecture

The BSA disconnects SodaBotL agent programs from low-level system interaction, details of networked communications, and privileged or unsafe operations. Figure 1 shows the BSA architecture. The BSA consists of several interlinked processes, the heart of which is the BSA kernel which runs user applications in a controlled environment. The kernel runs all active SodaBotL agents while listening for incoming data on a number of channels. The top three circles represent listener processes which accept incoming TCP connections from other agents (and people). The listeners are necessary to allow incoming connections while the kernel is busy and therefore unable to read incoming data.

SodaBot uses a dedicated Netscape browser for its user-interface. (There is also a TK-based user-interface not shown.) The Netscape browser connects to the BSA kernel via a high-priority port to permit quick response to user input.

The BSA also periodically checks for new e-mail, which may either be for its owner or for the BSA itself. In order to deal with network downage and network firewalls, BSAs can fully communicate through e-mail. Users can also utilize the SodaBotL Mail agent to act as a mail filter.

SodaBot also has several ways of interacting with pre-existing applications. SodaBotL has an an expect ([Libes, 1990]) dialog mechanism for "talking" to other interactive programs. There are also a variety of primitives for connecting to other networked applications. (These will be discussed in section 4.1.)

3.1 Naming

SodaBot implements its own naming scheme on top of the standard internet protocol. Each BSA has a name which resembles an e-mail address, e.g., foo@bar.com, but the user and host names don't need to correspond to internet-addressable entities. BSAs are interconnected via a three tier server hierarchy shown in figure 2. *Site servers* provide contact information for all the BSAs running at a particular site (or group of sites), and *global servers* are used by the site servers to locate other site servers. For example, to contact *foo@bar.com*, we would first have to contact *server@bar.com*, which would then provide *foo*'s contact information. If we didn't know how to contact server@bar.com, we would have to query the global servers to locate it. This name resolution process is invisible to agent programmers, as is the error handling for dealing with a BSA which moves between our contacts with it.

4 SodaBotL

SodaBotL is a very high level programming language for creating software agent applications. It offers sophisticated primitives and control structures which simpify the creation of complex, distributed programs, and it frees programmers from dealing with low-level details of TCP/IP, system calls, sockets, mail servers, X-windows, etc.

The SodaBot system was not intended as a universal computational platform. For example, when writing a meeting scheduling agent, it is quite in-line with the SodaBot philosophy to interface with an external scheduling program and not rewrite one in SodaBotL. We believe agents should capitalize whenever practical on all available resources whether they are people, pre-existing applications or other agents. SodaBot makes it easy for agents to connect to all three.

SodaBotL programs have three types of procedures: (1) a main procedure which is run when the agent is started; (2) *requests*, which are things that other agents can ask this agent to do; and (3) subroutines, for the local use of the agent's own code. For example, figure 3 contains the skeleton of a Merchant agent in an agent-based commerce system we have developed. The procedure labelled with the word *Agent* is run when an instance of the agent is first started. The requests *Quote*, *Sell*, and *Sales_volume* are things that other agents can ask this merchant agent to report about. Finally, *Check_inventory* and *Determine_price* are subroutines used internally by the agent.

SodaBotL has a syntax modeled loosely on TCL and C. As in TCL, SodaBot primitives are frequenly multi-word, with the first word naming a general class of commands, the second naming a method in that class, etc. Scalar variable references are prefixed with a $, lists with a @, and hashtables with a %.

4.1 Primitives

One of the main things SodaBot agents do is interact with other agents. SodaBotL includes a wide-range of primitives, the foremost of which are shown in figure 5, for directly supporting these interactions. This and the following sections will examine some of the more useful of these and other SodaBotL commands.

An agent can declare itself a member of some group so that other agents can locate it. For example, in our commerce simulation, merchants can declare their merchant status via:

```
Agent member ("merchants") -global;
```

A *buyer* agent could then identify the self-identified merchants via:

```
@sellers = Agent members? ("merchants") -global;
```

The variable *@sellers* would now contain a list of the merchants BSAs. The `-global` flags specify that this data should be distributed/retrieved from a global rather than a local server. This is essentially the same distinction as referring to "merchants in my area" vs. "merchants in the world."

The buyer agent could query the merchants about the price of some product with:

```
($price, %data) =
  Agent Query Merchant::Quote("camera") -user @sellers;
```

The *Merchant::Quote* argument directs the query to the *Quote* request of the *Merchant* agent.

4.2 Thread Management

SodaBot's thread management primitives are contained in figure 6. We need to use them in the previous example because presumably there is more than one merchant BSA. Therefore, the buyer agent will receive multiple replies to its query. SodaBot creates a new thread in the buyer agent as each of these replies arrive, and each thread continues running at the statement following the *Agent Query*. A simple way to gather data from commands which create multiple threads is to use the *Collect* primitive. It assembles the data received by each thread into shared data structures and then terminates the thread. Afterwards, the parent thread resumes at the subsequent statement.

```
Collect (@price, %data) =
  Agent Query Merchant::Quote("camera") -user @sellers;
Thread resume -thread_count length(@sellers)/2;

foreach $bsa (@sellers) {
  if (defined @price[$bsa])
    print "From $bsa, I received price @price[$bsa]\n";
}
```

The *Thread resume* command tells *Collect* to resume running after half of the merchants have responded. We could also manage the threads manually, at the price of slightly more complexity, if we wanted the buyer agent for example to keep a real-time updating display of current prices:

```
($price, %data) =
  Agent Query merchant::get_quote ("camera") -users @sellers;
$form = make_new_form($form_template, $price, %data);
Web Query $form;
$merchant = %web["agent"];
Agent Request merchant::sell ($product) -user %data[$merchant][address];
```

As each new price is received, the buyer agent uses a (not shown) *make_new_form* subroutine to update an HTML form listing all responses received so far. This form is displayed in the user's Web browser by the *Web Query* primitive which also has the side-effect of halting the current thread. Figure 4 contains a portion of the form displayed by the browser. After the user makes a selection in the form, the parent thread is restarted at the next line. The input variables defined in the HTML form are directly available to the program via the *%web* hashtable. The *Agent Request* primitive is used to run the *sell* request on the BSA of the merchant the user selected.

4.3 More Networked Code

There are several ways other than *Agent Query* and *Agent Request* to begin a computation on a BSA, two of which are discussed here. To start a new instance of an agent on someone's BSA, we use:

```
Agent Start $agent -user $person ;
```

where *$agent* and *$person* are variables containing the agent and BSA names respectively. Omitting the *-user* keyword causes the agent to be started on the local BSA.

To move an agent and its state to another machine, we could use the *Agent Hop* primitive; after an *Agent Hop*, the agent program continues running at the next statement but on a different BSA. For example, this snippet collects users' bookmarks files:

```
Require file access "~/.netscape/bookmarks.html";
@bsas = Agent members?("web experiment");

foreach $bsa (@bsas) {
  Agent Hop $bsa;
  %bookmarks[$bsa] = File read "~/.netscape/bookmarks.html";
}
Agent Hop home;
```

This code travels around to all the BSAs which are members of *web experiment* and stores their bookmarks files in the *%bookmarks* hashtable. Each iteration of the foreach loop occurs on a different BSA. At each of these BSAs, the code runs as the BSA's owner, so the tilde (~) symbol in the filenames are interpretted with respect to that individual.

Although SodaBot distributes the actual agent program during *Agent Hop*s only if the destination BSA has never run that agent before, this code is not as efficient as simply querying each BSA for it's bookmark file, because the agent's ever-growing local state must always be transmitted. Regardless, we have yet to come across any good reason for using mobile computations. We always prefer using *agent query* and *agent request*, particularly because SodaBot automatically distributes SodaBotL agent source code among BSAs as is necessary. (See section 4.5.) This issue is reminiscent of the debate between using goto vs subroutine calls in many programming languages. The distinction is a matter of programming style, not of expressive power; there is nothing one can accomplish which the other cannot. However, it seems invaluable to have very well-defined abstraction barriers in complex, distributed agents, and these barriers are very much broken by having individual computations move around the network. We prefer the alternative offered by SodaBot – to have the source code move around the network and to have agents remotely issue calls to it.

4.4 Security

The above code explicitly declares a single user file that it requires access to. In general, agents can request an entire class of capabilities or they can explicitly list specific privleges they require. SodaBot's notion of security is that agents should be secure both in function and in time, i.e. agents should only be allowed to do certain things and they shouldn't spend excessive doing them. All SodaBotL agents are run inside a safe environment which limits their operations and how long they can run based on their privilege level. Figure 8 contains the SodaBotL privlege request primitives. Agents that are deemed safe can be compiled and run automatically when they are received by a BSA. Otherwise, the user-customizable *receive_new_agent* agent determines what to do with them.

Agent running time is counted only when an agent is actively running on a BSA. For example, consider an agent which fetches the HTML source of a Web document:

```
Web fetch "http://www.timeinc.com/time/";
```

After connecting to the server, the BSA schedules other activities until the server is ready to transmit the requested data which can sometimes take a substantial amount of time, particularly with HTTPD servers. The BSA reads the incoming data as it becomes available and never blocks waiting to read from the server. Therefore, the agent doing the fetch is not penalized for time it doesn't actually get to use, and the BSA doesn't waste time waiting for data to become available. The BSA will also try re-establishing the connection several times if a remote agent or server is too busy or disconnects prematurely.

4.5 Automated distribution

By now, we are familiar with SodaBotL primitives such as

```
Agent Notify weather::alert($forecast) -users @weather_watchers;
```

This command contacts the BSAs in *@weather_watchers* and tells them to run the *alert* request of the *weather* agent.[4] What we have not addressed so far is how this *weather* agent becomes available to these BSAs in the first place.

[4]The *Agent notify* command differs from *Agent request* in that it is non-blocking and doesn't accept return values.

When an agent is created on a BSA, a copy of that agent is usually sent to the BSA's site server. The site server can then distribute the agent to other BSAs upon request. When a BSA receives a request for an agent it doesn't have, it will try to obtain that agent from the requester's site server and from any self-declared software agent repositories. These repositories run on global servers and stay updated with the latest agent releases. Agent authors can specify to some degree how and if they want an agent distributed with the *Set agent dist* primitive in SodaBotL. For example, an agent can be declared private and non-distributable by SodaBot system software by:

```
Set agent dist private;
```

Such agents (or their authors) must arrange for their own distribution before they are run.

Agent names can also specify where to obtain the agent from, which is in effect an extension of the agent naming scheme that makes agent names globally unique. For example, we could address a particular implementation of the weather agent via:

```
Agent Notify weather::alert::bar.com($forecast)
  -users @weather_watchers;
```

This specifies that the weather agent should be obtained from the site server at bar.com, namely *server@bar.com*. The specification could also include the actual BSA to obtain the agent from:

```
Agent Notify weather::alert::foo@bar.com($forecast)
  -users @weather_watchers;
```

A BSA doesn't usually request an entire agent. Rather, it asks for a part of the agent, for example, the *alert* request of the *weather* agent. Agent programs are distributed in sections called *groups* which are (possibly overlapping) subsets of the agent program. Each request can include itself in one or more groups:

`Request` *name* (*args*) {*groups A, B, C*} {...}

When a BSA wants to obtain this request, all the other requests which are members of its groups are sent along as well, as are all required subroutines. For example, in figure 9, if a BSA asked for request *req 2* which is in groups A and B, it would receive their other members also, along with all their invoked subroutines.

Groups let an agent be divided into sections that have different distributions. This is useful both for efficiency (in terms of prefetching relevant code) and to give an agent's author control over where sections of the agent's program are distributed.

5 An Application

We are currently using SodaBot to build an agent-based "brain" for the MIT AI Lab's Intelligent Room. The Intelligent Room project explores advanced human-computer interaction and collaboration technologies. The objective of this project is to create an intelligent room capable of interpreting and augmenting activity occurring within it.

A scenario that has been implemented and tested using SodaBot has someone in the Room pointing at a city on a displayed map and saying, "Show me the weather here." In order to do this, the Room needs to: (1) recognize who spoke and what they said; (2) determine where they were pointing at the time they spoke; (3) fetch the requested information; and (4) display it on a monitor near their current location.[5]

The Room's architecture involves a hierarchy of agents which interconnect the Room's sensors and effectors and link to external information sources. Low-level agents interact directly with Room hardware (e.g. cameras) and software applications, while intermediate agents reconcile and consume the information they gather.

We view building this brain as an excellent test of SodaBot's efficacy, because the Room's novel environment allows us to evaluate SodaBot as an AI research platform according the criteria of both [Brooks, 1991] and [Etzioni, 1993]. The Room's brain is "physically" grounded in the real-world. The Room's cameras and microphones allow agents to deal with the type of complex, unpredictable and genuine phenomena that Brooks argues is essential for a core AI research testbed. However, the agents also process abstract, symbolic information that actually represents something extant, thereby satisfying Etzioni's desiderata for core AI research. For example, when a user asks about the weather, the Room needs to do more then recognize a *weather* token — it needs to get that information and display it to the user. Agents can do this using a variety of information retrieval systems available on the World Wide Web, primarily our lab's START system ([Katz 1990]).

We present several examples of agents connecting with the Room, starting with the person locating agent from step (4) in the scenario given above. We then examine two other agents which build on top of this one.

Suppose we would like to locate a person in the Room with respect to an imaginary X-Y axis on the Room's floor. To do this, we collect data from the Room's camera systems which state where the person was with respect to each camera's image coordinates. We then need to reconcile this data to determine where the person was actually located in Room space coordinates. This reconciliation is performed by a separate C-language based neural net

[5]Ideally, this monitor would also need to be viewable with respect to their current gaze direction.

application which has been previously trained on the current camera set-up.

Each camera is connected to a C-language program which performs multiple-person tracking and interfaces to a BSA via its STDIN/STDOUT. The BSAs keep track of observed events (e.g., the movement of individual's bounding boxes) and the time they occurred and can both automatically notify other agents about new events and respond to their queries about previously occurring ones.

The position agent is shown in figure 10. When first started, the agent connects to the Room-space coordinate generating program via SodaBot's *expect* mechanism. After the *location* request is queried, it uses *collect* to gather information from the BSAs connected to the Room's cameras. The *thread resume* command insures that the data collection doesn't take more than 0.25 seconds. With this collected data, it then calls the *reconcile_cameras* subroutine which prepares and sends data to the reconciliation program started by *expect*. The output of this program causes *expect* to run *Agent Reply* which responds with the coordinates originally requested.

Figure 11 contains a more general-purpose location procedure which is part of the *user* agent. This request attempts to locate its BSA's owner: (1) on the BSA's host machine; (2) in the Intelligent Room; and (3) via the finger facility on the local mailer (which reports current login information).

Finally, figure 12 shows part of the agent which interfaces with one of our speech recognition systems via a TCP server. The *listen* request is run whenever new data is received. If the statement begins with the word *determine*, the rest of the words are sent to a natural language based, information retrieval system available over the World Wide Web. The text pulled out of the retrieved HTML document is then spoken through a speech synthesizer which was connected to the agent through an external library. If the spoken statement is of the form "Where is *person*?" the agent asks that person's BSA for his current location using the agent given above. For example, we can ask "Where is David?" and have the speech synthesizer reply "David is in the Intelligent Room."

Finally, if the user asked about the weather in some city, the agent retrieves that document from a commercial weather service, and after removing the advertisements from the HTML form, displays it in the user's Web browser.

6 Conclusions

In this paper, we have described a system for creating complex, distributed software agents. In our applications, the SodaBot system has proven to be an extremely useful tool for dealing with the dynamic, distributed, and heterogeneous nature of both the computational and real worlds.

We have recently begun to incorporate a planning language into SodaBotL to provide the agents with greater reasoning and coordination capabilities. Other future research directions include automated reasoning about agent programs and creating a distributed information retrieval system.

References

[AAAI, 1994] *Working Notes of the AAAI Spring Symposium on Software Agents*, Stanford, CA. 1994.

[Brooks, 1991] Brooks, Rodney. Intelligent without representation. *Artificial Intelligence*, 47:139-159. 1991.

[CACM, 1994] *CACM - Special Issue on Intelligent Agents*. 37:7. 1994.

[Cardelli, 1994] Cardelli, Luca. Obliq: A language with distributed scope. Digital White Paper, Systems Reseach Center, Digital Equipment Corporation, 1994.

[Etzioni *et al.*, 1992] Etzioni, Oren; Hanks, Steve; Weld, Daniel; Draper, Denise; Lesh, Neal; and Williamson, Mike. An approach to planning with incomplete information. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning, KR-92*, Cambridge, MA. p115-125. 1992.

[Etzioni, 1993] Etzioni, Oren. Intelligence without Robots: A Reply to Brooks. In *AI Magazine*, Winter, 1993.

[Gray, 1995] Gray, Robert. Transportable Agents. Ph.D. Thesis Proposal. Dartmouth College, NH. 1995.

[Katz 1990] Katz, Boris. Using English for Indexing and Retrieving, In Artificial Intelligence at MIT: Expanding Frontiers, edited by P.H. Winston with S.A. Shellard, vol.1, ch. 6, MIT Press, Cambridge, MA. 1990.

[Kautz *et al.*, 1994] Kautz, Henry; Selman, Bart; Coen, Michael; Ketchpel, Steven. An experiment in the design of software agents. *Proceedings of the Twelfth National Conference on Artificial Intelligence, AAAI-94*, Seattle, WA. 1994.

[Libes, 1990] Libes, Don. expect: Curing Those Uncontrollable Fits of Interactivity, in *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, California, 1990.

[Sun, 1995] The Java Language Specification. Beta Draft. Sun Microsystems White Paper, Sun Microsystems Inc., 1995.

[White, 1994] White, James. Telescript technology: An introduction to the language. General Magic White Paper, General Magic Inc., 1995.

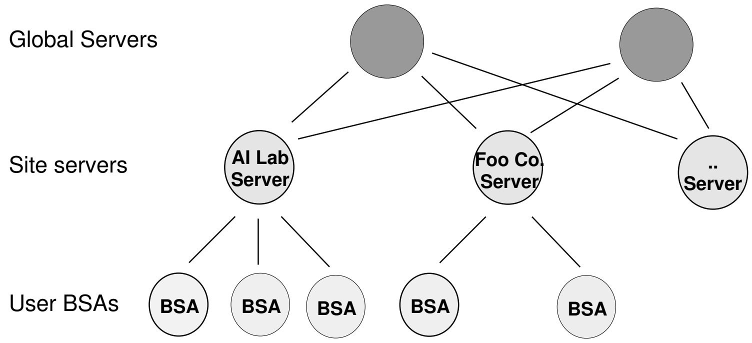

Figure 2: The server hierarchy

```
Agent Merchant () {...}

Request Quote ($product) {...}
Request Sell ($product) {...}
Request Sales_volume (@dates) {...}

Sub Check_inventory ($product) {...}
Sub Determine_price ($customer, %buying_history) {...}
```

Figure 3: The types of procedures in a SodaBotL program

Figure 4: A view from the browser

Command arguments listofBSAs	Remote function on/regarding given BSAs
Agent Start *agent* ...	Starts given SodaBot agent
Agent Hop *BSA*	Transfers current computation
Agent Fork ...	Transfers copy of the current computation
Agent Connect ...	Opens maintained connections
Agent Query ...	Asks for information
Agent Notify ...	Notifies of condition
Agent Request ...	Asks to run an agent request
Agent Member(?) ...	Declare/Query about BSA group membership
Agent Exists ...	Determine if BSAs exists
Agent Busy? ...	Return work-load measure
Agent Online? ...	Is BSA currently accessible?
Agent Refuse ...	Refuse request from another BSA
Agent Reply *args*	Return values to requesting BSA

Figure 5: Inter-agent communication primitives

Command arguments	Local function
Simul $\{\mathrm{Comm}_1 \ldots \mathrm{Comm}_n\}$	Perform communications simultaneously
Collect *arrays Comm prim*	Coalesces received data into named arrays
Thread Merge ...	Merges multiple threads
Thread Resume ...	Resumes waiting thread at given condition

Figure 6: Thread management primitives

Command arguments	Local function
Web Load *url*	Sends Web browser to location
Web Fetch *HTML_url*	Returns source of named url
Web Display *variable*	Displays HTML source contained in *variable*
Web Query *variable*	Displays HTML form contained in *variable*
Server Listen *port, request*	Runs TCP server on *port*
Server Disconnect	Closes current server connection
Server Shutdown	Stops server
Client Connect *host, port*	Connects to TCP server
Client Send ...	Send to client
Client Read Line ...	Read from client
Client Read Regexp ...	Read multiline reg expr from client
Client Disconnect ...	Close connection
Library *file, language*	Dynamically load Perl5 or C-language library
Import *function_list*	Link to functions in library
Expect ...	See text

Figure 7: Additional connection primitives

Command arguments	Gives access to
Class user agent	user's files
Class system agent	anything user could do from shell
Class library agent	system SodaBot libraries
Require system access *access_list*	named executables
Require file access *access_list*	named files
Require library access *access_list*	named libraries
Require time *time_limit*	real time *time_limit*

Figure 8: Resource request primitives

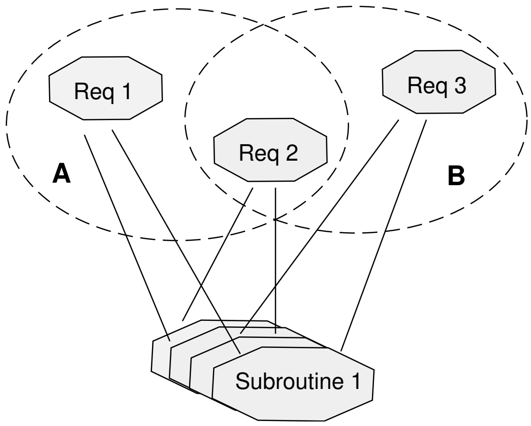

Figure 9: Groups of requests

```
@cameras = ("left1@hci.room", "right1@hci.room");

Agent position () {
  $number = '\d+\.\d+';
  Agent member ("position");

  # Run Room-space coordinate generating program
  Expect "cd /home/rh1/tlp/hci; track-nnet" -background {
    "Track NNET started" <main> <loop>;
    "($number) ($number) ($number)\n" <loop> {Agent Reply $1, $2, $3;}
  }
}

Request location ($person, $time) {
  Collect (@x_1, @y_1, @x_2, @y_2)
    Agent Query camera::position($person, $time) -users @cameras;
  Thread resume -timeout "0.25 seconds";

  reconcile_cameras(@x_1, @y_1, @x_2, @y_2);
}

Sub reconcile_cameras (@x_1, @y_1, @x_2, @y_2) {
  foreach $x_1 (@x_1) {
    $y_1 = shift @y_1;
    $x_2 = shift @x_2;
    $y_2 = shift @y_2;
    $string .= "$x_1 $y_1 $x_2 $y_2";
  }

  Expect send "$string\n";
}
```

Figure 10: The Position Agent for locating people in the Intelligent Room

```
Agent user () {...}

Request location () {
  if (idle() tlt "5 minutes")
    Agent reply ("at console of $hostname");

  $position_bsa = Agent member? ("position");
  ($x, $y, $z)  = Agent Query position::location($me, time())
                   -user $position_bsa;
  if ($x) Agent reply ("in the Intelligent Room");

  Expect "finger $user_email" {
    "User\s+Real Name\s+What\s+Idle\s+TTY\s+Host\s+Console Location"
      <main> <loop>;
    "$user_login\s+$user_name\s+\w+\s+(\S+)" <loop>
      Agent reply "at console of $1";
    "$user_login\s+$user_name\s+($time)\w+\s+(\S+)" <loop>
      if ($time tlt "5 minutes") Agent reply "at console of $1";
    "Plan:" <loop> {Expect quit;}
  }

  Agent Reply 0;
}
```

Figure 11: A request for locating a person

```
Library "~sodabot/Lib/say.pl";
Import say;
Include "~sodabot/Lib/user_list";
Include "~sodabot/Lib/city_codes";

Agent speech () {
  Server listen 3004 listen;
}

Request listen ($text) {
  ($command, @words) = words($text);

  if ($command eq "determine") {
    $form = web fetch "http://sakharov.ai.mit.edu/Start-Question?@words";
    $text = textify HTML $form;
    say($text);
  }
  elsif (match($text, "where is (.+)")) {
    $bsa = %user_list[$1];
    $location = Agent query user::location() -user $bsa;
    $location ? say("$1 is $location") : say("I don't know where $1 is");
  }
  elsif (match($text, "what is the weather in (.+)")) {
    if (! $city = %city_code[$1])
      say("I don't know about city $1");
    else {
      $doc = Web fetch "http://www.intellicast.com/weather/$city";
      # Remove ads from HTML document
      $forecast  = match $doc,
                     '<A HREF="#pagetop"><IMG SRC=[^>]+>(.+)\$'
                     -multiline;
      Web display $forecast;
    }
  }
}
```

Figure 12: An agent for connecting to a speech recognition system