Sneezy Note #1: The Sneezy Manifesto

Lars T. Hansen

November 8, 1995; February 21, 1996

Abstract

Wouldn't it be nice if dynamic programming tools like debuggers, profilers, and visualizers could just be downloaded and put to work right away? With this as my basis, I present a manifesto for a portable, low-level program interaction API called *Sneezy*[1] and outline its structure and use. Other Sneezy Notes contain the details.

1 Manifesto

1.1 The need for portable run-time tools

Tools which need to interact with running program are very useful and therefore deserve to be supported. Examples include debuggers, profilers, visualizers, and trace systems. Various research groups have implemented these tools (often several times), and it would be nice if a user could simply download a tool from the net and apply it to her current program.

1.2 The need for a common interface

The low-level interface to the program deals with breakpoints, memory modification, signals, events, and the like. It is typically very run-time system dependent. If each run-time system has a different interface, each tool must be ported to each new run-time system, for a total of $O(n \times m)$ ports for n run-time systems and m tools.[2] In contrast, by providing a common, portable interface to a running program and simply porting this interface for each new run-time system, we reduce the porting effort to n ports: one port for each run-time system.[3] Sneezy is intended to be such a common interface.

1.3 The need for an agent-based interface

What is really needed is an agent-based interface. By this we mean that the *parallel program* is under control of an *agent*, and that the *client* (the tool) interacts with the agent at all times to have its requests performed. An agent-based interface can be contrasted with the more common interface where the tool must have direct access to the program's address space, either via the Unix `ptrace` system call (implying that the tool and the program is running on the same machine) or through shared memory (implying either that the program cooperates with the tool or that the tool is linked into the program's executable). The direct approach is not very flexible because of its requirements and because it becomes truly cumbersome in a program that has multiple address spaces, sometimes on multiple physical machines. The agent model, however, allows the use of `ptrace`, shared memory, or shared address spaces for efficiency without actually requiring it (communication can be over TCP sockets, raw ATM, native message passing, or anything else the agent supports, and the client and agent can be on separate machines), and since the client communicates only with the agent, the physical structure of the computation remains hidden to the clients that do not wish to discover it (assuming that the agent is indeed willing to divulge it).

[1] "Scheme Nerd's Breezy", because I'm a Scheme nerd and because the basis for the work is a similar but more restricted toolkit called Breezy, developed by Darryl Brown at the University of Oregon.

[2] Of course, sets of tools will usually abstract the interface away, but if you collect a number of individual tools from different research efforts, the problem is real.

[3] Note also that n is (much) smaller than m.

1.4 The need to keep it simple

In providing the interface we can take any number of approaches, the extremes of which I will call "maximal" and "minimal". The minimal approach provides only very basic support, and the client must work fairly hard to accomplish anything. The maximal interface is typically very language-specific and provides high-level operations, making the client typically very simple.

For example, the agent can provide an interface to read data from the program's address space. In a minimal interface, the client must provide an address and the number of bytes to read; the interface is on the machine level. In the maximal interface, the client must provide some programming-language context in the form of an expression to evaluate, and data will be returned in some easily decodable type-dependent form. In the minimal interface, the client needs to manipulate the symbol table and the run-time context somehow in order to get to the data; in the maximal interface the symbol-table manipulation is part of the agent implementation.

Event filtering is another example. Not all events are interesting to all clients, so clients will want events to be filtered. In the minimal interface, events are filtered at the client exclusively. In the maximal interface, there will be sophisticated event-filtering functionality in the agent, accessible through the interface, probably based on some programming language which gets compiled on the fly for efficient filtering. Intermediate solutions will provide simpler mechanisms, like turning events on and off under program control.

It is my view that the Sneezy agent should be as simple as is reasonable, that is, it should approach the minimal view. There are two main reasons. The first reason is that of tractability: the interface cannot be expected to provide all the functionality that every client will ever want, and this seems especially true for parallel programs. The second reason is one of discovery (or science): by putting almost all functionality in the client, we can discover what the performance problems of this approach are, and find new, minimalist solutions to them, subsequently augmenting the agent interface. Doing it this way, we have a fighting chance of learning something about what makes the interface efficient, rather than simply including every feature and just observing that the resulting interface is "fast" or "convenient".

By making the agent simple, much complexity is pushed into the client, but this complexity can be conquered by providing libraries which implement common tasks and idioms. Since the low-level interface is portable, these libraries will be portable, and effort can be spent on making them efficient.

2 Overall design

2.1 How it should work

Abstractly, we can think of a Sneezy-based system as having three parts, as outlined above: the *agent* is an entity which provides interfaces in the form of two APIs to the *client* and to the *parallel program*, where the client is a program which is run to control the parallel program, for example a debugger. (In the following the parallel program will be referred to simply as "the program".) The program runs as a number of *threads* in a number of *contexts* (address spaces); the number of threads in a context does not need to be constant at run-time, nor is it necessary for there to be the same number of threads in all contexts.

Initially the program is ready to run but not running; it is under control of the agent from the very start. The agent waits for a client to connect to it. When the client connects, it receives information about the program and then gets to control it via the agent. The client can now instrument the program by enabling event handlers, setting breakpoints, initializing data in the program's address space, and so on. These actions are performed by sending *commands* to the agent. The client then starts the program, also by sending a command to the agent.

The program runs until it one of its threads hits a breakpoint or an event point for an enabled event. At this point, the thread is halted and control over the thread (and the control over the thread's address space, although not over the other threads in that address space) is given to the client; the mechanism is that the agent sends an *event* to the client and waits for further instructions from the client. The other threads in the address space and any other threads in other address spaces are still computing.

When the client has control over the thread, it can send more commands to the agent to enable more events, disable previously enabled events, read data from or write data to the thread's address space, and so on. When the client is done dealing with the event it sends a command to the agent which will cause the agent to continue the execution of the thread.

2.2 The events

Typical events will be: tread terminates, barrier entry, barrier exit, remote fetch, remote put, remote service request, service remote service request, begin parallel section (for data-parallel languages like pC++ and HPF), create distributed data structure, delete distributed data structure, enter function, and exit function. There will also be an event to signal that a thread has reached a user-defined break-point. Most likely there will be some events specific to certain languages and run-time-systems.

2.3 The commands

Typical commands will be: continue thread, terminate thread, read thread's data, write thread's data, enable event for thread, disable event for thread, call function in thread's context. There might also need to be functions to retrieve information about distributed data structures at run-time, and similar dynamic run-time-system parameters, if they cannot be implemented on the client side.

2.4 Event filtering support

It will probably be useful for some event filtering to be done in the agent rather than in the client, especially if the agent is "close" to the program and the client is not (as when a part of the agent runs in each context but the client does not). The forms of event filtering we have envisioned so far come in three flavors: event counters, implicit continue, and parasite programs.

Filtering by event counting is a straightforward idea: step thread through $n - 1$ events of this kind; signal only the nth event. To implement this we associate a counter with the event in the agent; the cost in complexity and code is minimal.

"Implicit continue" is an event attribute which causes the thread to continue execution immediately while the event is delivered asynchronously to the client. Implicit continuing can be combined with event counting to create events which are sent to the client but which only causes the program to stop after n instances of the event.

A parasite program is a procedure which is inserted into the program at run-time in such a way that when an event is triggered, rather than sending the event to the client, the parasite is invoked with parameters indicating the nature of the event. The parasite can determine whether the event would be interesting to the client and if so, will have access to a mechanism which allows it to send the event to the client. The parasite can have (per-thread) state and can therefore perform meaningful event filtering. The thought is that parasites will be written sometimes before debugging starts, and sometimes during a session for ephemeral purposes.

2.5 Agent interfaces

There will be two agent APIs: one for the client side, and one for the program side.

The program side API is a collection of functions which a thread will call to signal events, one function for each event, and in addition some housekeeping functions. Whoever implements the instrumentation of the parallel program and the instrumentation in the run-time system will only need to deal with these interfaces.

The client side API is another collection of functions plus a number of numeric constants which define event types, commands, and so on. There will be functions to install event handlers, remove handlers, read data, write data, get thread information, and continue and terminate threads, one for each command. A particular concern is that the parallel program's primitive data types may not be representable as primitive data types in the client, so primitive program data will always be represented as abstract data types in the client, complicating the interface somewhat.

2.6 Multiple clients

In some cases it is useful to be able to connect multiple clients to the agent, so that the program is under multiple-client control. An example of such a case is when a replay debugger controls the execution of a program while a state-based debugger lets the user manipulate and inspect the program's state. The replay debugger must be in control in order to run threads according to the replay log, but the state-based debugger must be in control to single-step the program. Another example is having a visualizer at the same time as a state-based debugger, or multiple visualizers to visualize multiple data structures. Execution monitors like load-balancing tools can also be connected while other tools are present.

We are still working on designing the multi-client support, so the following paragraph is a sketch of the current state of affairs. See note #5 for more details.

The model currently adopted by Sneezy centralizes control in the agent. Clients are fairly independent and interact mostly independently with the agent. Each client enables its own events and installs its own breakpoints. One client is the *master* and is allowed to control the execution of the program; it transfers the master property to one of the other clients (each of which, when it is not the master, is a *slave*) via Sneezy, so Sneezy keeps track of who gets to control the program. This model is desirable in that different clients can communicate with the agent using different protocols (for example, a replay manager can be linked into the program and will therefore be fairly efficient, whereas a state-based debugger can be communicating via shared memory or over sockets). In addition, the mode for control transfer is standardized and makes it simpler (although not trivial) to integrate multiple-client tools from different sources. It is problematic that a multi-client agent is rather more complicated than a single-client one, and not nearly as efficient; however, it is possible to make multi-client agents pay-as-you-go.

3 Implementation issues

3.1 Performance, performance, performance

If Sneezy is going to be used for non-trivial tasks of non-interactive nature (e.g. tracing, on-line profiling, visualization, lightweight instrumentation by parasites, and replay) then good performance is extremely important. For a non-blocking event to be much more expensive than one or two procedure calls (discounting the cost of any actual processing performed by the event handler) is probably unacceptable in several of these instances. It would seem that the above architecture does not allow this low cost to be obtained, in that the nature of Sneezy is typefied as two asynchronously computing threads communicating with messages. There are, however, optimizations to be made which can reduce the cost to an acceptable level. I will outline some of these optimizations without regard to architectural appropriateness; not all methods are acceptable on all machines.

3.2 Communication specialization

The communication layer in Sneezy can detect (sometimes with a little help) whether the communication can be specialized for performance. For example, a client and a program running on the same physical machine can communicate via shared memory. A client linked into the program can communicate with the program using simple procedure calls (although there may be some extra requirements placed on the client in this case). A client running on a machine connected to the parallel machine by a fast network like a memory channel or ATM can use the network, if the client application's structure warrants it. If the client and program are running on machines with the same or similar architecture (in terms of byte order, word size, and floating-point representation) they can communicate using raw binary data rather than a portable representation.

The specializations are transparent to the client code due to the structure of the APIs. Even if the client connects to the agent over a portable-representation socket the communication layer can detect faster channels if available.

In addition, we believe that Sneezy can be implemented on top of CORBA and similar distributed-object paradigms, and we fully intend to ensure that this remains the case. Implementations can then reuse existing infrastructure (which may be optimized for the host system).

3.3 Semantic restrictions

Some clients can be fast if they obey certain restrictions. For example, if each event handler is a procedure which performs some simple processing and then returns to its caller, and the client is linked into the program's address space, then the handler procedure can be called directly as the handler for a nonblocking event, with very high performance (cost of 1 or 2 procedure calls, depending on how the comm layer is structured).

It might be interesting to discover ways of communicating to the agent that a client obeys certain rules.

3.4 Parasites

Computation can be moved from the client and into the agent or program by installing parasites which perform the computation without the event loop being entered; for complex filtering or condition checking, this would be a win, performance-wise.

4 Related Projects

Some projects and proposals which are relevant or related to Sneezy are:

- The performance instrumentation server proposed by Douglas Pase.
  See `http://www.cs.wisc.edu/~pase/server_proposal.html`.
- Dynascope, from Griffith University, Australia.
  See `http://www.cit.gu.edu.au/~sosic/dynascope.html`.
- The OMIS work at Technische Universität München.
  See `http://wwwbode.informatik.tu-muenchen.de/~omis/`.
- The *Breezy* system from the University of Oregon.
  See `http://www.cs.uoregon.edu/paracomp/tau/breezy/index.html`.
- The `hpd` debugger project at Los Alamos National Laboratories.
  See `http://www-c8.lanl.gov/dist_comp2/hdd.html` [sic] but that link seems to be broken.
  Perhaps try `http://www-c8.lanl.gov/dist_comp2/devel_tools.html` instead for a higher-level link.
- The `p2d2` debugger project at NASA.
  See `http://www.nas.nasa.gov/NAS/Tools/Projects/P2D2/`.
- The Super Symbol Table work at the University of Wisconsin.
  See `http://www.cs.wisc.edu/~paradyn/CSTOtalk`.
- Paradyn, from the University of Wisconsin.
  See `http://www.cs.wisc.edu/~paradyn`.
- Parasight, which was done at Encore Computer Corp. by Aral and Gertner; there's a paper in the 1989 Workshop on Parallel and Distributed Debugging.