Sneezy Note #3: Sneezy: the Protocol, Version 1

Lars T. Hansen Darryl I. Brown

June 21, 1995; October 10, 1995; February 21, 1996

Abstract

This document describes the protocol for a single client in its interaction with the Sneezy agent. The specification is preliminary; see the last section for a list of known shortcomings.

1 Scope and definitions

This document defines the protocol for Sneezy on the level of interaction between the debugging client (the program doing the debugging, known as the *debugger*) and the debugging server (the program being debugged, known as the *debuggee*). Sometimes it is useful to distinguish between two parts of the debuggee: the *user program* and the *agent*: whatever was added to the user program in the form of instrumentation and debugger support when the program was compiled for debugging.

For the time being we define events meaningful in the current version of `pC++` augmented with a subset of the `HPC++` computational model, defined below.

Communication is performed end-to-end on a channel between the debugger and the debuggee; we don't care how this is implemented (sockets, specialized networks (ATM), shared memory). We are interested in defining the contents of the data at the endpoints of the channel. The channel is defined to be reliable and preserve the ordering of messages in the sense that if two messages A and B are put onto the channel at one end in that order, they arrive at the other end in that order.

For concreteness we also specify one particular format for the data as seen by the endpoint at the debugger side of the channel. This specification is merely one mapping of the abstract data format into a concrete representation and should not be considered definitive.

2 Data format

For simplicity (it's a prototype) all communication is with textual representations, and all messages can be parsed as (nested) Scheme lists using Scheme's `read` procedure. A list is consists of a left parenthesis followed by an arbitrary number of data elements (including lists) followed by a right parenthesis. All non-list data are either decimal signed integers, or quoted strings following the syntactic rules of Scheme [1, 2].[1] In summary, a string is a sequence of characters between two double quotes (ASCII value 34) where literal double quotes in the string are represented by a backslash (ASCII value 108) followed by a double quote, and where literal backslashes are represented by two consecutive backslashes. There is no standard provision for non-printable data, although most implementations seem to handle such data well.

Some entities used in the rest of the document are defined here.

Context ID: A nonnegative integer (abbreviated `CID`) in the range $0 \ldots n-1$ where n is the number of contexts.

Thread ID: A nonnegative integer (abbreviated `TID`) in the range $0 \ldots m_i - 1$ where m_i is the number of threads in context i. Note that in HPC++ with dynamic threads, m_i is not constant, and not all thread IDs are valid at all times.

[1] We may want to allow floating-point numbers, but on the other hand, maybe they are better encoded in hex/text.

Global Pointer: In `pC++`, a list of two elements where the first element is a context ID and the second element is an address valid within that context.

Barrier ID: In `pC++`, a nonnegative integer starting with 0 which denotes the dynamic sequence number of a barrier (all barriers are global).

Collection ID: In `pC++`, a nonnegative integer denoting a particular collection. The ID is valid across all contexts.

3 Computational Model

Our computational model is a subset of the computational model of `HPC++`. It is compatible with the computational model of `pC++` in the sense that any `pC++` program which does not violate the `pC++` computational model (by depending on processor number or other external side effects) will be a valid program in our computational model.

A computation consists of one or more *contexts*, each of which contains one or more *threads* of computation. One of the threads in each context (the thread identified as thread 0) is known as the *distinguished thread*; it runs the `main` program in that context. The other threads, if there are any, participate in data-parallel work but are otherwise idle. Each context has a unique `CID`, and each thread has a unique `TID` within its context.

The model has the following storage classes: global, top-level, static, auto, and register. The register and auto classes are as they are in normal `C++` programs: allocated to registers and on the stack, local to each thread. Top-level and static variables exist in one copy in each context, and include heap-allocated storage obtained from `new`, `malloc()`, `sbrk()` and so on. Modifying a location which has top-level/static storage class makes the modification visible in that context only. Global storage lives in an address space which is distributed across the contexts. Global objects are allocated with a special storage allocator (called $\mathtt{new}_{\mathtt{G}}$) which returns a *global pointer* to the object. A global pointer can be operated on in restricted ways outlined in the `HPC++` white paper [3].

A barrier is always a global event: all threads participate in a barrier.

4 Interaction Model

Abstractly, there is one communication channel which connects the debugger and the debuggee. As outlined in section 1, the present document considers this channel to be abstract.

In general, when a program is being debugged it happens in the following manner.

1. The program is compiled for debugging, which will cause it to be instrumented and linked with debugging support. The nature of this instrumentation and support is unspecified in this document; it can be anything from nothing (as when the agent is actually a separate process setting breakpoints in the user program with the aid of a traditional debugging mechanism like `ptrace()`) to massive instrumentation and linked-in support (as in the case of the current `pC++` system).

2. The debuggee is started (either by the programmer or some master program, like for example the debugger itself). The debuggee initializes itself and, knowing it will be debugged, waits for the debugger to contact it by creating a channel endpoint at some well-known address and waiting for a connection to that endpoint.

3. The debugger connects to the debuggee by connecting to the debugger's endpoint address.

4. The debuggee sends some vital program information to the debugger. These data are defined in section 5.

5. Each context in the debuggee sends a `context-ready` message to the debugger.

6. The debugger waits for all the `context-ready` messages. When they have all arrived, it knows that the distinguished thread of each context is in the debugging command loop, and normal debugging can commence.
7. While a thread is in its command loop, the debugger sends commands to it to install or remove handlers, inspect or modify variables, and so on (commands are defined in section 6.1). When the debugger has performed the tasks it wishes to perform, it sends the `continue` command to the blocked thread, and that thread resumes execution and executes until it gets to an even for which a handler is installed, at which point it passes the event to the debugger and goes into the command loop again.
8. When the distinguished thread of a context terminates, then the context stops executing, and a `thread-terminate` event is sent to the debugger. The debugger can now inspect the state of the context, but cannot run it further. To terminate the context completely, the debugger sends a `terminate` command to it. The context then truly stops executing and will no longer interact with the debugger.
9. If all the contexts terminate (i.e. receive a `terminate` command), then the entire program should terminate (the process(es) should exit).

5 Startup Data

The startup information is a structure:

```
( <version-number> ( ( <context-number> <number-of-threads> ) ... ) )
```

that is, a list consisting of a version number (a nonnegative integer) and an association list where the key is the context number and the datum is the number of threads in that context.[2]

6 Communication

Communications are either debuggee-to-debugger (events and replies) or debugger-to-debuggee (commands). The formats for the communications are in principle dissimilar and are presented separately in the following.

6.1 Debugger-to-debuggee messages

A debugger-to-debuggee message is a *command*; it has the following format:

```
( <command> <context-id> <thread-id> <data> )
```

where the `<command>` is a decimal numeric code and the `<data>` depends on the command. The commands and their codes are:

Cmd#	Command name
0	`continue`
1	`terminate`
2	`install-handler`
3	`uninstall-handler`
4	`read-data`
5	`write-data`
9	`call-user-function`

`Continue` is sent to a thread to make it continue its execution. The command loop of the thread will be unresponsive until after the next time the thread has sent an event to the debugger.

Data: An empty list.

[2] It's not a real association list, the elements are lists rather than pairs, in keeping with the general data format.

`Terminate` is sent to a thread to make it halt execution. The thread's command loop will no longer respond to commands, and no more events will be forthcoming from the thread. If the thread is the last live thread in its context, the context may also terminate.

Data: An empty list.

`Install-handler` installs an event code for a given event type in a thread. Any previously installed handler is quietly superseded. The next time the thread encounters an event of the specified type, an event containing the specified event code will be sent to the debugger, and the thread will wait in its command loop.

Data: A list constaining an `<event-type>` and an `<event-id>`.

`Uninstall-handler` removes any installed handler for the given event type, such that the next time the thread encounters an event of the given type, no event is sent to the debugger, and the thread will not halt its execution. If there is no installed handler for the given type in the thread, the command acts as a no-op.

Data: A list containing an `<event-type>`.

`Read-data` retrieves data from the thread.

The debugger sends a global pointer and a byte count to the debuggee and receives the data located at that location back; the data can then be decoded by the debugger.

Data: List containing a context ID, a pointer valid in that context, and a byte count.

Reply: The reply is a datum encoded as described in section 6.2.

Note: It is unclear what it means to send a read request to a context when the context ID in the global pointer is not that of the context receiving the request. We leave this behaviour undefined, but stipulate that if such a request is honored, then the correct data must be returned. Also, there is a worry about whether the data read has been read in a consistent state.

`Write-data` replaces some of the running program's data with new values.

Data: A list consisting of a context ID, a pointer valid in that context, and encoded data as described in section 6.2.

Note: Write-data must worry about synchronizing updates to shared data structures so that data appears consistent to the application at all times.

`call-user-function` calls a function in the running program.

Data: A list containing a context ID, a function pointer valid in that context, and a set of parameters. The parameter structure encoding is not currently defined.

Reply: A datum in the standard encoding (possibly).

Actions not covered by the preceding list of commands include: introspection facilities, stack backtrace and access, symbol table access, setting and resetting breakpoints for single-stepping, nonblocking events, and agent-side event filtering. Many of these issues are discussed in section 7.

6.2 Data Encoding

The standard encoding consists of a length field followed by encoded bytes of data. The length field is a two-byte big-endian number encoded as a hexadecimal number using two characters for each byte; for example, the number 4660 becomes the four characters `1234`. The bytes are currently encoded using two characters for each byte and a hexadecimal encoding. This is not efficient, but simple. Bear in mind that binary encodings are possible and desirable in settings other than that of S-expressions.

In a message, the entire encoding is wrapped in a string datum. For example, the string `FOO` becomes is encoded as `0003464f4f`.

6.3 Event types

An `<event-type>` is one of the following:[3]

Type	Name	Class
0	`error`	all
1	`thread-terminate`	main
4	`begin-barrier`	all
5	`end-barrier`	all
6	`begin-fetch`	all
7	`end-fetch`	all
8	`begin-put`	all
9	`end-put`	all
10	`begin-rpc`	all
11	`end-rpc`	all
12	`begin-parallel`	main
13	`enter-element`	all
14	`exit-element`	all
15	`begin-service-rpc`	all
16	`end-service-rpc`	all
17	`user-event`	all
18	`begin-create-collection`	main
19	`end-create-collection`	main
20	`begin-delete-collection`	main
21	`end-delete-collection`	main
22	`end-parallel`	all
23	`enter-function`	all
24	`exit-function`	all

"Class" is either "main" (distinguished thread only) or "all" (all threads participate).

`Error` is sent when the running program or its agent detects an error, for example an error in a command sent to the program or an internal error. Internal errors can be unexpected termination as a result of a fault (segmentation fault, bus error, abort, divide-by-zero, and so on).

Data: a list consisting of a numeric error identifier (as yet not defined).

Notes: see section 7 for a discussion of how errors can be handled. The preceding description is incomplete.

`Thread-terminate` is sent when a thread in the debuggee terminates. Since the debugger knows (or can know) how many threads run in each context, there is no need for a separate context-terminates message as such. If the context terminates abnormally, an `error` event should be sent.

Data: An empty list.

`Begin-barrier` is sent when a thread enters a barrier. If a `continue` command is given, the thread will truly enter the barrier and will be unresponsive until all threads have entered the barrier and they all send an `end-barrier` event.

Data: A list consisting of the barrier ID.

Notes: Would it also be useful to have a reference to program point here, or can that be figured out from the stack backtrace?

`End-barrier` is sent when a thread leaves a barrier.

Data: A list consisting of the barrier ID.

[3] There needs to be an event which deals with exceptions/`longjmp`, as such events can really mess up the debugger's view of the debuggee.

`Begin-fetch` is sent when a thread dereferences a global pointer. In the current version of pC++ this occurs only when a fetch of a different collection element is performed (a `Get_Element()`). In pC++ version 2 and in HPC++, any fetch through a global pointer will cause this event to be sent.

Data: a list containing the source address (global pointer pointer broken out as context and pointer), the destination address (ditto), and the number of bytes.

`End-fetch` is sent when the dereference is done.

Data: As for `begin-fetch`.

`Begin-put` is sent when a thread performs an assignment through a global pointer, whether the element is stored locally or not.

Data: a list containing the source address (global pointer pointer broken out as context and pointer), the destination address (ditto), and the number of bytes.

`End-put` is sent when the assignment through the global pointer is finished.

Data: As for `begin-put`.

`Begin-rpc` is sent when a thread starts a remote service request in another context.

Data: A list containing a global pointer to the parameter block (broken out), the length of the parameter block, a global pointer to the collection element which is the target of the request, and the request code.

`End-rpc` is sent when a thread finishes the remote service request.

Data: As for `begin-rpc`

`Begin-service-rpc` is sent when a thread services a remote service request from a thread in another context.

Data: Not defined.

`End-service-rpc` is sent when a thread has finished servicing a remote service request from a thread in another context.

Data: Not defined.

`Begin-parallel` is sent by the distinguished thread just before it goes into a parallel section.

Data: An empty list.

`End-parallel` is sent by the distinguished thread after all slave threads have finished data-parallel work in a parallel section, and the distinguished thread is about to go into a sequential section.

Data: An empty list.

`Enter-element` is sent by a thread when it is about to begin processing of a collection element in a parallel section.

Data: A list containing a global pointer to the element, broken out.

`Exit-element` is set by a thread when it has finished processing a collection element in a parallel section.

Data: As for `enter-element`.

`User-event` is sent when a user event is executed in the program.

Data: A list containing an event code, a global pointer to additional data (broken out), and the length of the data.

`Begin-create-collection` is sent when the thread is about to start creating a collection.

Data: A list consisting of the collection ID, if known, or -1 if not.

`End-create-collection` is sent when the thread has finished creating a collection.

Data: A list consisting of the collection ID.

`Begin-delete-collection` is sent when the thread starts destroying a collection.

Data: A list consisting of the collection ID.

`End-delete-collection` is sent when the thread stops destroying a collection.

Data: A list consisting of the collection ID.

`Enter-function` is sent when the thread enters a function.

Data: A list containing the function's symbol table ID and a time stamp.[4]

`Exit-function` is sent when the thread leaves a function.

Data: As for `enter-function`.

6.4 Debuggee-to-debugger messages

A message from the debuggee to the debugger is either an event notification or a reply to a command from the debugger. These messages are encoded in the same way, with a reply represented as a special kind of event. The format is a four-element list:

```
( <event-id> <context> <thread> <data> )
```

where `<event-id>`, `<context>`, and `<thread>` are decimal nonnegative integers and `<data>` is a Scheme datum whose format depends on the `<event-id>`, as explained below. The `<event-id>` is the numeric value of the installed event handler or a reserved event ID from the following list:

0	`context-ready`
2	`reply`
3-100	(reserved)

7 Other things

A number of issues are unresolved. Several are mentioned earlier. Some others are:

- *Timestamps.* There should be a configuration option settable (somewhere) which would cause a processor-local timestamp to be added as the fifth element of every event message. The timestamp should be a single number expressing the running time, in milliseconds, since the start of the program. (or microseconds? We'd overflow pretty quickly in the C code on a 32-bit box.)

- *Non-blocking events.* A couple of methods have been considered for moving some intelligence into the server to avoid some of the overhead of blocking for each event sent to the client.

  One method is to flag events as non-blocking: before continuing a thread, flag all events as non-blocking that you do not want to stop at, and set up some other event to block. For example, at the time of a begin-create-collection, all barriers should be nonblocking and only end-create-collection should block. A nonblocking event would be sent to the debugger, but the thread would not wait for a reply–the continue is implicit.

  Another method is to have non-blocking counting events: a counter is associated with each event in each thread, and decremented for each event of that type. When it gets to 0, it blocks. Otherwise, it just

[4] We need to formalize exactly what it means to pass a symbol table ID. For now the idea I'm getting at is that for every program symbol (that is, a combination of name, signature, and scope) there is a unique identifier which will allow us to find information about the symbol in the symbol table, and that this identifier can be known early enough to use in instrumentation. It is possible that this ID should simply be the address of the function, but that causes trouble for member functions.

sends the event and continues. This is similar to pure nonblocking events but gives the opportunity to 'step' efficiently at the expense of having to occasionally deal with event counters that reach 0 (when you don't want to). This method has a further sophistication: the counting, non-blocking, non-messaging event. This is like a counting non-blocking event, but while the counter is non-zero, events are not sent to the debugger (but the counter is still decremented).

- *Event filtering.* There can be a fair bit of overhead associated with passing all the events to the client. Some mechanisms for filtering out events in the agent might be useful (that is, more useful than simply turning events off and on). Two methods that have been suggested are regular expressions and downloadable code. The latter is potentially much more powerful, but even regexps can be nifty.

  As an example of what one can do with regular expressions, consider a regular expression language where an event is a tuple in angle brackets (with some conventient short names, say) and a mechanism for argument extraction. We can the create a filtering event by saying something like this:

  ```
  (filter-event "<beg-cre-coll>[^<end-cre-coll>]*<end-cre-coll,$id>"
                "<my-event-id,$id>")
  ```

  where `my-event-id` is just some unique event ID. The above expression represents a filter which takes a `begin-create-collection`, any random events, and `end-create-collection`, and bunches them up into a single event representing a collection creation. Context and thread IDs are implicit.

  Sneezy note #4 talks more about event filtering.

- *Dynamic profiling.* If function entry and exit events are timestamped, we can do dynamic profiling. Even if they are not timestamped, we can do a dynamic call-graph type of display.

- *Replies and errors.* Should all commands have replies? If so, why? If not, why, and which commands should have replies? How are errors reported if not thru a reply? If errors are reported through a reply, then the error cannot be detected unless all commands have a reply, as the debugger won't know when to look for an error reply. Another option is out-of-band data. A third option is to communicate not directly on the socket but on queues, and having reader and writer processes in the client which check incoming messages and signal errors. This simulates out-of-band data.

- *Events for servicing fetch and put.* Should we have these? I think they are needed for replay analysis, but then, will Sneezy be used for replay analysis?

- *Debugger-initiated fetch and put.* There are some problems here, as outlined in the notes for the `read-data` command, having to do with how other threads (not necessarily blocked) react to read requests from foreign threads. It seems that for good general support (remember, the debugger is under program control, so saying that "no-one would do that interactively anyway" is not good enough) we need active messages or frequent polling. Needs discussion.

8 Shortcomings

The following is a list of missing features, problems with the current specification, and so on.

- *Reading and writing registers.* The debugger will need this functionality.

- *Machine information.* It should be possible for the client to request data about the specifics of the hardware of the target machine.

- *Attach/detach.* There is no way for a client to attach to an already running computation, nor is there a way to detach from one, and the semantics of client termination are not specified.

- *Real breakpoints.* There are currently no commands to set or remove breakpoints in the code.

- *Protocol specification.* The protocol should be specified more abstractly, not in terms of s-expressions.

- *Call-user-function.* This mechanism is weakly specified.
- *Atomic command groups.* OMIS has a notion of atomic event groups. It may be that we should consider them for Sneezy to avoid certain race conditions.

References

[1] IEEE Standard for Scheme.

[2] Revised^4 Report on Algorithmic Language Scheme.

[3] HPC++ White Paper.