Sneezy Note #5: Proposal for Multiclient Support

Lars T. Hansen

February 25, 1996

Abstract

The present paper outlines the protocol used for multi-client management and control. It should be considered preliminary.

The proposal is much more complicated than I like, and the clients that use this proposal will have to work a bit to maintain things, but I don't know how to make it simpler and still have it be general and as free from restrictions as I'd like to. As pointed out below, one solution might be to support multiple models of varying complexity.

The mechanism for event selection is believed to be implementable but should be specified rather more carefully.

1 Introduction

Supporting multiple clients fundamentally conflicts with Sneezy's philosophy of simplicity and minimalism. Multiple clients create a number of problems for the agent implementation and for the clients themselves.

Multiple clients will need to be aware of each other; there does not appear to be any general and reasonable way in which multiple clients can co-exists independently *for other than simple applications* (monitoring, say). The most complicated case seen so far is the interaction between coexisting state-based and event-based debuggers. On the other hand, simple applications can make do with much less hair than the current specification has, and it is possible that multiple mechanisms should be available, depending on the needs of the client.

The main problem is contention for control after an event has been passed to multiple clients. The only way that I know of to solve this problem is to impose an arbitration process on the clients. The arbitration process has as its goal to make sure that only one client gets control when an event is signalled, but at the same time, that something meaningful happens in the clients which do not receive control. The two questions that must be answered are, how is arbitration done, and, what happens to clients which do not receive control?

Multiple clients can connect to an agent in three distinct manners. In the simplest case, there is never more than one client connected: the clients communicate among themselves and connect and disconnect by some protocol outside Sneezy. From Sneezy's point of view, this model is desirable, because it keeps the agent very simple: arbitration is trivial. There's no difference between single-client and multi-client support. However, the model makes it hard to implement clients which all see all events efficiently, and there will probably be a tremendous amount of protocol among clients. I call this the "A-or-B" model.

A second model connects a single client to the agent, and other agents connect to the first client, which then performs event forwarding. The clients again communicate among themselves about control without involving Sneezy. Again, Sneezy remains unchanged, but every group of clients is forced to implement their own multi-client protocols, and they are all likely to be different, so even if clients are aware of each other, they are likely to be incompatible. Furthermore, all clients depend on the interface of the first client to the agent, so even if some secondary clients could use a fast (same-machine or same-address-space) interface to the agent, they will be running over a slow channel if the primary client is. Standard libraries can alleviate these problems somewhat, and the model is not necessarily undesirable. I call this the "A-then-B" model.

The third model 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. I call this the "A-and-B" model. The arbitration method is non-trivial and must allow policy choices to be made by the clients. Sneezy has adopted this model because it allows most aspects of the two others to be simulated, thereby avoiding pre-emption of design choices for clients as far as possible.

The A-and-B model is desirable also 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). It is problematic that an A-and-B multi-client agent is rather more complicated than a single-client one, and not nearly as efficient; however, it should be possible to make multi-client agents pay-as-you-go.

2 Arbitration of Control

I propose that arbitration is implemented by designating a *master* client for each thread in the parallel program, that is, for each thread, exactly one client is its master at any one point in time. When the thread signals an event, control of the thread is always passed to its master client. Mastership is on a per-thread basis because other solutions (per-context, per-program) seem to be too coarse for most clients. Clients which are not the master of a thread are *slaves* relative to the thread. Slaves can receive event messages but do not automatically receive control along with the event.

There are mechanisms for passing mastership to other clients and for receiving mastership.

Each client needs to keep track of its own status relative to a thread.

3 Event Semantics in Multiple-client Situations

3.1 Semantics

The proposed event semantics in multiple-client situations are low-level, but general.

In order to make the discussion somewhat precise I introduce some notation. Let E denote an event type and T a thread. Each thread T has a master client, denoted by M_T. Slave clients with respect to a thread T are denoted by S_{iT}. The location of an instruction in the code of the parallel program is denoted by L. For each location, there is a set, possibly empty, of distinct event handlers H_L, where each member of H_L is a pair (E, C), signifying that the client C has installed a handler for event E.[1]

When a T reaches an L for which H_L is nonempty, let

$$\begin{aligned} ME &= \{(E_i, M_T) \in H_L\} \\ SE &= H_L - ME, \end{aligned}$$

that is, ME is the set of event handlers for the thread's master client, and SE is the set of event handlers for the thread's slave clients.

There are two cases: ME is nonempty, or ME is empty (in which case SE is necessarily nonempty).

If ME is not empty, then one of its members is selected and signalled to M_T in the usual manner; M_T receives control of the thread. (The method of selection is important; see below.) The other members of ME and all the members of SE are remembered by the agent. That is, M_T receives one event and may later gain access to the other events for the same thread and location. M_T must handle the event and decide what to do about the others; such policy questions are outside the scope of this document but mechanisms will be defined below which allow the additional events to be read from the agent, and which allow events to be sent to other clients.

If ME is empty, then the meta-event `foreign-event` is sent to M_T (along with control of the thread), and all members of SE are remembered in the agent. Again, M_T must decide how to deal with these events.

If any of the events in ME or SE are flagged as non-blocking, they are dispatched and removed from the sets before any blocking events are handled by the above policy.

[1] I ignore parasites for now, but the model extends readily to them.

3.2 Event Selection

One problem remains unadressed, namely that of data connected with an event. Many non-breakpoint Sneezy events have data associated with them (beyond the location at which the event happened). Sometimes these data are generated by the running application. If events are "remembered" as outlined above, how are the data for multiple events retrieved from the agent or program?

We make one important, but reasonable assumption: *apart from breakpoints, which are unrestricted, there may only be one non-breakpoint Sneezy event at a given location in a thread at any point in time.* For example, there may not be both a `begin-create-collection` event and a `user-event` event at the same location. There may be a `user-event` and one or more breakpoint events (parasitic or not), or no non-breakpoint events but multiple breakpoint events.

An important consequence of the above assumption is that if the agent sets up events by installing breakpoints in the parallel program, then there will be restrictions on how events may be installed. If the program is instrumented at compile-time, we can end up with code like this at the beginning of a function which creates a local collection:

```
snz_enter_function_event( ... );
snz_begin_create_collection_event( ... );
...
```

where the events are at the same position in the original source but at different locations in the program, by the previous definition.

However, if breakpoints are used to set up non-breakpoint events, the events will end up at the same location and will therefore not fit our protocol as defined above.

The observation about how events coexist and the given restriction lets us define an event selection order which always works: if there is a pre-defined Sneezy event that carries data to be signalled, then select it first and pass its data along. Otherwise, select any data-less event.

4 Command Semantics in Multiple-client Situations

The most important restriction on commands is that *slaves cannot issue commands which affect the program.* This includes installing and removing handlers, reading and writing data, and so on. Any command which requires interaction with a thread in the program cannot be issued by a slave client.

The reason for the restriction is that slaves are asynchronous with respect to the behaviour of the thread.

In an effort to ease this restriction we could allow handlers to be installed and removed by slaves, but under the restriction that the commands may not take effect until the next time the program stops at a previously enabled event (enabled by the slave or any other client in that thread), but I'm reluctant to do so until we have some experience; such an easing would complicate the agent.

5 Extended Startup Semantics

Multi-client agents have a different startup sequence than single-client agents: startup is more complicated.

The first client connects to a multi-client-aware agent. The client receives the startup data as usual and then a `context-ready` from each context. The client is the master for all threads and need so far do nothing different from a client connected to a single-client agent. The client may discover its ID by issuing the `get-client-id` command to the agent.[2]

The second client connects. It receives the startup information followed by a single `slave-ready`; the client must go into slave mode. Since no event handlers are installed for program events, no program events will be received. When `slave-ready` has been sent to the new client, `new-client` events are sent to all previously existing clients, containing the ID of the new client.

There are no provisions for keeping track of which clients are which. The obvious solution is for the first client to start other clients in a known order, keeping track of the new IDs and maintaining an internal table

[2] `get-client-id` should be supported in single-client-agents as well, returning -1, say.

mapping IDs to client names. Using the supplied messaging mechanism, clients can communicate among themselves to share information and for other purposes.

Typically, clients will be started in a setup phase and the primary client will temporarily transfer control to new clients to allow them to install handlers for desired events.

6 Problems

Are there deadlock problems?

7 Rationale (needs work)

There are numerous problems that have to be addressed when multiple clients are present. In the following, assume that A and B are clients and that A is the master w.r.t. thread T.

Scenario 1: Thread T signals event E, which both A and B have requested be delivered. Since A is the master, E is clearly delivered to A, and A receives the control over T.

1. Is E delivered to B?
2. If E is delivered to B, is there any way for B to gain control over T, even momentarily?
3. If E is delivered to B and B gets to gain control, what happens to A while B is in control?
4. If E is delivered to B, does B get to gain control if A changes the state of the program (the program counter, say) while it is in control?

Scenario 2: Thread T signals events E1 and E2 (which are installed at the same location). A has requested E1 and B has requested E2.

1. Is E2 delivered to B, and if so, with what semantics?
2. Is A aware of E2, or B aware of E1, and if so, how are they made aware of them?

Scenario 3: Thread T signals events E1 and E2 (which are installed at the same location). A has requested E1 and B has requested E2. E1 and E2 are breakpoint events with associated code (parasites).

1. Is E2 delivered to B?
2. What the parasites get to do to the code (inspect, modify, change PC, call code which may cause events to get triggered, trigger events themselves)?

8 Event Definitions

`(slave-ready -1 -1 (client-id ...))` is sent to the new client when it connects (after the startup message) if there are other clients present. The client must respond by going into slave mode. The data consists of the IDs of all existing clients, not including the new one. This event may not be ignored.

`(new-client -1 client-id ())` is sent to existing clients when a new client connects. Clients connect in a strictly sequential (but arbitrary) order, so this message makes sense. The client ID is the ID of the new client. When existing clients receive this event, the new client is up and running and and in slave mode.

`(become-master -1 new-master-id (old-master-id cid tid))` is sent to a client to make it the master for the the thread. The client must go into master mode for the thread. If the `tid` is -1 then the client should become master for all threads in the context. If both the `cid` and the `tid` are -1, then the client should become master for all threads. This event may not be ignored.

(`foreign-event cid tid (eid data)`) is sent to the master client for a thread when there are events for other clients but not for the master. The event passed as an argument is the event which carries data, if any; if there is no event with data then some data-less event is passed, and the `data` field is the empty list.

(`message -1 client-id string`) relays a message from the given client. This event may not be ignored.

9 Command Definitions

(`get-client-id -1 -1 ()`) returns the ID of the calling client. The return value is a single nonnegative integer.

This command can be issued at any time; it is directed at the agent.

(`get-program-state -1 -1 ()`) returns the state of the program: for each thread in each context there is a flag which indicates whether the thread is in the command loop. The information is correct at the time it was gathered, and all events are delayed until the information has been delivered, so it can be considered to be current at the time of arrival.

The reply is structured as a list of lists. Each sublist has three elements: the context id, the thread ID, and a flag. The flag is the state of the thread: 0 means in command loop, 1 means running, 2 means blocked. It is not always possible to distinguish between 1 and 2, and as far as the client is concerned, there is no practical difference between them.

This command can be issued at any time; it is directed at the agent.

(`transfer-master -1 client-id (cid tid)`) gives mastership of the named thread to the given client. If the client ID is not valid, an error event will be sent with code `SNZX_TXR_BADCLIENT`. If the `tid` and `cid` are -1 then mastership of all threads owned by the client are transferred to the new client.

This command can be issued by the master at any time.

(`sendmsg -1 client-id string`) sends the message to the given client; the message will be received as a `message` event (q.v.). If the client-id is -1 then the message will be sent to all clients (excluding the originator).

This command can be issued at any time. If the recipient does not exist an error event will be generated with code `SNZX_SEND_BADCLIENT`.

(`auxiliary-events -1 -1 (cid tid)`) requests a list of event information about the given thread. The thread must be stopped, an event must have been delivered, and the requester must be the master of the thread. The reply is a list of (event-type, client-ID) pairs. The context, thread IDs, and data are known.

(`signal-event -1 client-id (eid cid tid data)`) sends the event to the given client. The eid and the data should have been obtained by a call to `auxiliary-events`. This can be sent by any client at any time.

10 Unreolved issues

These are some unresolved issues above and beyond the implementation grunge.

- What happens when a client exits? If the client still has mastership over some threads, they will suddenly find themselves master-less, and events will be unhandled. We can't simply turn off all events for the thread, because valid slave threads may exist. Perhaps we should simply transfer mastership to some arbitrary slave, along with an indication that the master has died.
- There is a race condition in the pair

```
(transfer-master -1 x ...)
(signal-event -1 x ...)
```

because other events can be sent to the new master between these two unless the thread in question is stopped. We might want to implement a mechanism whereby commands can be grouped into an atomic unit.