Sneezy Note #7:
Sneezy Client API for Multi-threaded Scheme

Lars Thomas Hansen

February 15, 1996

1 Low-level Client Interface

1.1 Vocabulary

A *cid* is a Context ID. A *tid* is a Thread ID, relative to some *cid*. An *etype* is an event type, as defined in the protocol document (note #3). A *gptr* is a global pointer (a pair of context ID and address within that context).

1.2 Client Interface

The low-level client interface is defined by a module called `client`. Currently, only one connection is allowed at a time and the client module keeps internal state about the connection.[1]

(`client.initialize` *hostname portnumber ready-handler*) ⇒ *unspecified* *procedure*

The *hostname* is a string naming a machine (an IP address is not allowed), the *portnumber* an IP port number on that machine, and the *ready-handler* an event handler procedure (below). `client.initialize` attempts to connect to the named port on the machine and fails with an error if the connection cannot be opened. If the connection is opened, it initializes private data structures which will be used to service calls to other functions in the client interface.

(`client.cleanup`) ⇒ *unspecified* *procedure*

`client.cleanup` closes the connection to the agent.

(`client.data` *key . more*) ⇒ *data* *procedure*

The *key* is a symbol, one of `server-context-count`, `all-contexts`, and `threads-in-context`. The value of *more* depends on the value of *key*. `client.data` returns data about the parallel program the client is connected to.

If *key* is `server-context-count` then *more* should be the empty list and an integer denoting the number of contexts in the parallel program is returned. If *key* is `all-contexts`, then *more* should be the empty list and a list of all context numbers is returned. If *key* is `threads-in-context` then *more* should be a list containing the context number information is requested for, and the number of threads in that context is returned as a nonnegative integer.

(`client.control` *key*) ⇒ *data* *procedure*

Key is a symbol, one of `monitor`, `nomonitor`, and `peek`. `client.control` is a simple introspection facility for the client module (for developers). If *key* is `monitor`, then message monitoring is turned on, echoing all incoming and outgoing messages on standard output. If *key* is `nomonitor` then monitoring is turned off. If *key* is `peek` then a representation of the client module's internal state is returned.

`Client.control` can be called before `client.initialize` to turn on monitoring. Monitoring is off by default.

[1] This is a weakness and must be fixed.

(`client.continue` *cid tid*) ⇒ *unspecified* *procedure*

The command causes a `SNZC_CONTINUE` command to be sent to the thread. The effects are unspecified if the thread is not in its command loop or if the client is not the master of the thread.

(`client.terminate` *cid tid*) ⇒ *unspecified* *procedure*

The command causes a `SNZC_TERMINATE` command to be sent to the thread. The effects are unspecified if the thread is not in its command loop or if the client is not the master of the thread.

(`client.handle` *etype cid tid event-handler*) ⇒ *event-number* *procedure*

The *event-handler* is an event handler procedure (below); it is installed as the event handler for the named event for the given context and thread. The *event-number* is the internal event ID selected by the client module. The effects are unspecified if the thread is not in its command loop or if the client is not the master of the thread.

(`client.unhandle` *etype cid tid*) ⇒ *unspecified* *procedure*

The installed event handler for the named event, context, and thread is removed. The effects are unspecified if the thread is not in its command loop or if the client is not the master of the thread.

(`client.read` *cid tid gptr len*) ⇒ *string* *procedure*

The *gptr* names a location in the parallel program. `client.read` reads data starting at that location for *len* bytes and returns the data as a string. The returned string is a standard encoding (including length) of the data read; the standard encoding is discussed in the protocol document.

The effects are unspecified if the thread is not in its command loop or if the client is not the master of the thread.

Currently the client module supports only one outstanding request per context at a time, and if `client.read` *or other commands which have replies* are called multiple times per context (regardless of the threads at which the requests are directed) before the previous call has returned a reply, the requests will be serviced in an implementation-dependent order, and they will not overlap. One should therefore avoid dependencies on order and overlap, as deadlock is possible.

(`client.write` *cid tid gptr string*) ⇒ *unspecified* *procedure*

The *gptr* names a location in the parallel program, and the *string* holds a standard encoding (including length) of data to write into the program starting at the location named by *gptr*.

The effects are unspecified if the thread is not in its command loop or if the client is not the master of the thread.

1.3 Programming with the client package

An event handler procedure takes four arguments: event type, context ID, thread ID, and event data (a list). When an event is received, the event procedure is called, and no more events will be delivered until the event procedure returns.

Behind the scenes, the client runs as an independent thread which listens on the connection for incoming messages. Event procedures may therefore be called at any time. The client module protects all its data and no external synchronization is needed to call it.

There is a severe lack of good error handling: any error will crash the client module. This needs to be fixed.