CSpray Architecture

CSpray uses a streams based architecture. Streams are a connection based communication abstraction implementable in Unix System V Streams or TCP/IP sockets. Streams are separated into priority, information and control streams. The streams handle a continuous flow of information between application programs in the collaboration. We have worked with a variety of network level implementations including Ethernet and ATM (asynchronous transfer mode). By providing separate streams to tasks, we match the goals of heterogeneity of networks, platforms, and users by separating out different quality of service information.

Events are handled by message passing between collaborating remote hosts through the established stream connections. Events may be local and therefore non-collaborative or they may be collaborative. The latter type may or may not require floor control. Events are sent as variable length messages called `WriteGram`s and `ReadGram`s. Both message types have information on message length, message id, and sender.

For collaboration awareness, an internal module handles all messages with other collaborating hosts. It knows who it is connected to and how to transmit data to other collaborating programs. It also receives messages from collaborators and posts the relevant messages locally for the other modules of *CSpray*.

There are three procedures in the collaboration interface module: `InitCollab`, `GetFromRemote`, and `SendToRemote`.

`InitCollab` is called as soon as the main process is initialized. First, it initializes all structures relevant for collaboration. This involves filling in the header of the `WriteGram` structure with information specific to the local host. This information remains fixed in the `WriteGram` for the life of the process. A separate process is then invoked that blocks and listens for all collaboration events. When a collaboration request is made, this process informs the application via a stream.

All the collaborative input piped to the application is filtered by the `GetFromRemote` procedure. Here `ReadGram`s are parsed, and memory allocated to store incoming messages. This procedure is called at the beginning of the the applications's cyclic main event loop so that if this procedure returns with an event, all modules will have a chance to handle this event. For example, when an object is made sharable by a collaborator, the object module will process this event and create a new object. Since the object is generated remotely, it will copy the body of the `ReadGram` message into the new local object structure.

At the end of the main event loop, `SendToRemote` is called to allow events generated by any of the modules to be possibly broadcast to collaborators. For locally generated events meant for collaborators, a `WriteGram` is constructed and transmitted. If the event was not locally generated, for example when we just received a collaborator's new object event, `SendToRemote` deallocates the memory allocated for the `ReadGram` and returns.

Figure 1 shows the scenario where a new collaborator joins an existing session. Through an exchange of `WriteGram`s and `ReadGram`s, the Caller is updated with the current state of the session including any public cans and their associated visualizations, as well as eyecons of the other participants. The figure shows the exchange of new can information, eyecon information, and the ordering of events in time from the top to the bottom of the figure.

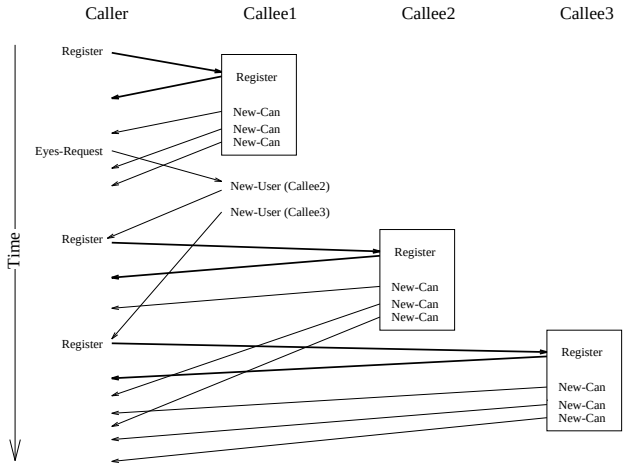

Figure 1: Illustration of process for a new caller joining a session with three participants that each have three public spray cans. The caller contacts Callee1 in order to join the session. Upon successful registration, the other two callee's (Callee2 and Callee3) are also registered with the caller. Information about public spray cans are also transferred at this point.