3. Simple Execution Model

This section describes a simple execution model for Joule. It is intended to describe the rules that all Joule computations must follow, and to give the reader an intuition of how Joule computations could actually get work done; it is not intended to represent an efficient implementation model. This simple execution model does not include resource management techniques or Domains (separately-executable pieces of code). Section 9.4 presents a more complete computational model after these concepts have been explained.

The execution model presented here owes much to the Actors execution model (**[2]**, **[65]**).

An executing Joule system consists of numerous servers sending messages to each other. Some of the servers, such as numbers, are *primitive servers,* built outside or underneath Joule; they include the basic servers from the kernel of Joule and foreign services provided externally. Execution bottoms out in these primitive servers. All other servers are *composite servers,* built in Joule from more primitive servers; they enact programmed flows of messages among primitive servers. To support the recursive abstraction of servers, primitive and composite servers have the same operational semantics so that clients cannot tell the difference.

All references to Joule servers are made via *ports.* Messages are never sent directly to a server, but rather to a port to that server. Each server can have multiple ports to it, each with a different behavior. These different behaviors are called *facets*.

The resource issue of who provides storage for messages will be dealt with in Section 9.4 in a future version of this manual. Strikingly, this particular kind of resource management problem is very similar to flow-control, which has been solved in the context of telecommunications: Tymnet and X.25 provide solutions, for example.

Messages sent to a port don't necessarily get to the server immediately. Instead, a *pending delivery* is made for the server receiving from that port. Execution proceeds by completing a pending delivery to the relevant server. The reception of the message by the receiving server and the ensuing computation in response to that message is called the *activation* of the receiving server.

Each composite server contains a collection of ports to other servers, and code to execute when activated with a message. The only ports accessible during the activation of a server by an incoming message are:

- the ports contained by the activated server
- the port to the incoming message
- ports to any servers created in the activation

The only actions a composite server can take when activated are:

- create new servers whose contained ports must be selected from the accessible ports

- send accessible port as messages to other accessible ports

These laws of computation restrict information flow among servers to message passing only; servers cannot, for example, compare ports for identity or side-affect global variables.

These laws do not directly provide the ability to change the collection of contained ports in the server; i.e., the server's state. The semantics of changeable state, so necessary for adequate modeling of real systems, is built in the language on top of the kernel semantics (and can be implemented efficiently). These abstractions (e.g., **Server** forms with **var** extensions) are described in Section 5.1.

When activated, primitive servers can perform arbitrary internal computation, so long as they respect these laws. They can contain ports to other servers, and can cooperate with each other (subject to the accessibility laws). They can change their internal state to include any other accessible servers. They cannot violate the modularity of the program by either reaching inside other servers (except cooperating primitive servers), or by referencing servers that were not explicitly accessible. Joule computation bottoms out by primitive servers cooperating with each other. For instance, integer addition bottoms out when the "+" operation is sent to a primitive integer with another primitive integer as its argument. The receiving integer gets the bits from the argument integer, computes a new result integer, and forwards the result channel to it; if the argument is not a primitive integer, then the receiving integer must send a message asking the argument to perform the addition.

The idiom of simple object-oriented message sending is as follows: the sending server, during some activation, creates a new tuple—the typical kind of server used for messages—and sends it to one of its other accessible ports. The delivery of that tuple is then pending for the server facet listening for messages on that port. When execution activates a server with the message, that server can then send to that message (considered as a tuple object). This allows it to extract the *operation* (the name of the tuple) and its arguments, for use in further computation. Because the tuple is a primitive server, when it receives messages to reveal internal parts of itself, it can do so immediately without spawning an infinite recursion of message sending.

An *operation,* in this execution model, is a unique token that can be compared with other tokens. They are not described in detail because they are replaced with public/private key pairs under the new regime described in the Energetic Secrets appendix.

Two other primitive server types, channels and arbiters, are used to interconnect servers into complex systems. Primitive Joule servers called *channels* have two facets, an acceptor and a distributor. The *acceptor* is for sending messages *through* the channel to other ports. The *distributor* is for controlling where messages sent to the acceptor get forwarded—messages sent to the distributor can forward the channel to other ports. The behavior of the channel is such that, for all messages sent to the acceptor port, a pending delivery of the message will be made for any port to which the distributor forwarded the channel.

To forward a channel is to forward all messages that have been or ever will be received on that channel.

Sending on the acceptor of the channel is equivalent to sending through the channel to each of the ports to which the channel is forwarded. This equivalence is *transparent:* sending to the acceptor of a channel is indistinguishable by the sender from sending directly to the ports of any servers to which that channel delivers. Messages sent through the channel are also preserved, so that if the distributor forwards the channel to any other ports, those ports will also get all the messages; a pending delivery to the new destinations will be made for every preserved message.

Channels have private access to *arbiters* for choosing among messages received. Arbiters are primitive servers that are not directly accessible at the programmer level; they are implicitly accessed using the `choose:` operation of distributors. Supplied with a port for results and a distributor containing messages, an arbiter chooses one of the distributor's messages and forwards all of the others to a newly-created channel. It then sends to the result port a message that contains the chosen message and the distributor to the new channel. Arbiters provide the fundamental non-deterministic choice required for synchronizing access to resources. For example, in trying to model a bank account, if two clients try to withdraw the entire balance, only one can get it. Arbiters are the selection mechanism for ordering requests to provide synchronization for servers.

The **If** form is built from Arbiters.

Channels and Arbiters and the programming techniques using them are described in Section 5.1.

Servers perform the same role as objects in object-oriented programming languages; however, they differ in that they are implicitly *concurrent, ubiquitous* (everything is a server), and *uniform* (all behavior is in response to messages). Because all behavior is in response to messages, and because messages wait until their recipient can respond to them, Joule inherently provides data-flow synchronization. Any server can send messages on a channel even before the channel has been forwarded to any other servers. When receiving servers are created, they respond to all the pending messages.

Because delivery of messages is not immediate, any delay in the creation of the receiving server is not apparent to the client.