Architecture overview

This is commentary for the Moose architecture slide, which was originally slide 34 of the WAMM-BAMM 2006 talk on MOOSE. This slide explains the organization of elements and information flow in MOOSE. Some terminology:
MOOSEGENESIS object element class object message message

This document will use MOOSE rather than GENESIS terms. Simulation entities (in yellow) These are MOOSE objects, typically representing computational entities. Examples:
- Molecules and reactions for a kinetic model
- Compartments and channels for a neuronal model Small white arrows between simulation entities

These are messages. They carry information between objects. In MOOSE they are like function calls from one object to another, which pass arguments that the target object uses to advance its calculations. However, messages are traversible: you can follow the arrows in either direction, to figure out how the simulation is wired up. Solvers

These take over the operations of groups of simulation objects. Examples would be a Hines Solver type object, taking over the compartments and channels that represent a single neuron. When such a takeover occurs, the affected objects become zombies. As far as any other object or part of MOOSE is concerned, the zombies are just the same. However, all operations on the zombies are actually referred to the solver. Solvers are meant to do specific calculations very fast, usually involving array-type and non-object oriented calculations. Solvers may also interact with the multiprocessor scheduler, or deal with their own threads. Multiprocessor scheduler

This ensures that all operations in the simulation are carried out in the right order. For simple simulations this means calling computation operations on each object for each time-step. As we involve the solvers this gets more interesting, because many solver algorithms use variable time-steps. When we spill over into multiple nodes, things get yet more involved, because we need to lump groups of data transfers together for efficiency, and ensure that all messages reach before we go on to the next time-step.

All this sounds pretty bad, but actually it is possible to modularize it to a large extent. The scheduler is actually just another set of objects communicating using messages. The calls to computation objects are themselves messages. Internode messaging

This is an MPI-based layer for forwarding messages between objects.

As far as the originating and target objects are concerned, these are just regular messages. What actually happens is that the messages go a ‹postmaster› object. The postmaster in coordination with the scheduler, takes the message arguments, lumps the data together, and sends this out to the target nodes. At the target node the data are reassembled in the target postmaster into ordinary messages that finally reach their target.

We still need to work out details of sending objects themselves back and forth. Some form of object serializing and converting this data into other messages will probably happen. The user interfaces

The GUI appears to the system as another bunch of objects sending and receiving messages. We're still working out whether to implement them in JAVA, FLTK, or something else. There is no particular need for the GUI objects to be on the same machine(s) as the simulation, or even on the same machine(s) as each other. The shell

This provides the core API for controlling objects: formation, assignment, calling functions, setting up messaging. It also keeps track of context, like current working object. This is the only part of the system that communicates with objects using something other than messages, though it uses those too.

Multiple shell objects are permitted. You could imagine a really big simulation being worked over by multiple people, like a patient etherized upon a table. The parsers

These provide scripting control over the simulation. There is of course the old Script Language Interface, imported from GENESIS. However, the shell is accessible through SWIG which means that most of the popular scripting languages could be used. Currently we have implemented Python as an alternate scripting language. External I/O

Lots of things happen here, but it boils down to getting information in and out of the system.
- Databases: These may be for model definitions, which go through special parsers, or for model results
- Model files likewise.
- SBW and other modeling systems communicate using XML formats for specifying models and data. Again, parser I/O.