Ryan Hughes [ryan@n-h.net]

Fault tolerant Python Clusters for parallel network-wide computation and simulation with Python Classes

Development aim:

Development of a **Python** runtime environment to distribute the application processing load of long and large tasks onto a large number of hosts that are connected by different networking options (LAN, WAN, email). It will have **transactional** components to handle and recover from disturbances (e.g. end of screensaver activity or idle-CPU processing) in the cluster network gently without interrupting the overall distributed task. The distribution inside the cluster should be invisible for the Python application modules and classes and their object instances. The complexity of scheduling and dispatching the execution of object instances should be controlled by the **Cluster Operating System**, which itself should be developed entirely in **Python**.

Brief description of the structure of the Cluster Operating System:

- On each host there is only a small amount of **long running Cluster Server Processes**, in most cases only one. Each cluster server process is identified by a network-wide server-process ID.
- Each cluster server maintains at least one permanent or semi-permanent **network connection** to another cluster Servers process.
- Communication between all participating components of the Cluster operating system and all components of its cluster server processes is done by **asynchronous messages**. Each cluster server can send a message to any other cluster server by passing through **one of the possible routes** to the destination.
- The main component of each cluster servers process is a central **message and task dispatcher** loop that connects itself by a pair of internal prioritized **message queues** to each participating component of it's process (1 input and 1 output). Each participating component inside the process is called a **Message Service** and is identified by a Message Service ID. Each cluster server process contains at least one of each of the following message services. The central message and task dispatcher loop takes care of distributing messages between the participating Message Services. Message Services are plug-ins taking care of the processing of messages to the corresponding services. Message Services receive messages by subscribing for certain Message Service types from the **message dispatcher**.
- Complex messaging transport Problems are solved by re-framing messages and processing them by a combination of **Messages Services.**

Message Services:

- **Object Message Service** hosts and manages a configurable amount of encapsulated objects dependent on the available system resources. The object message service translates incoming messages to object method calls and translates calls and changes made by the object to messages. It maps messages to python object properties and vice versa. A standard python application class is unaware that and how messages are being received or sent. The Object Message Service is in charge of creating, destroying and relocating objects to a new location in the cluster.

- **Name Message Service** resolves object location in the cluster according to their hierarchical oriented name, which is independent of the network structure of the cluster.
- **Connection Message Service** handles and maintains network connections to other cluster server processes (e.g. socket-based connections). It passes the message out of the processes memory to a different cluster server process on the same or on a different host.
- **Routing Message Service** passes incoming messages to the appropriate local message service or to a Connection Message Service if routing is required. Also forwards messages that are only a transit to a further connected cluster server process.
- **Rerouting Message Service** optional routing service for the case of network or failure of individual cluster server processes. Tries to transport messages along an alternative path to their destination.
- **Message Mirroring Services** (e.g. Persistence Services, that must be available all the time, and synchronized after partial failure)
- **Transaction Management Service** controls the execution and distribution of transaction, and handles exceptions that cause rollbacks of a transactions, and proper re-execution of the task or redistribution of the objects to a new cluster server. Used in conjunction with the Message Mirroring Service to guarantee the redundant completion of the execution step.
- **Binding Service** is a configurable mapping of objects' output properties to input properties of other objects in the network. For the runtime construction of applications out of pre-developed Python classes and components.

Application Development:

Distributed Python Applications are developed and implemented in the form of **Object Message Services** that are hosted by the **Cluster Server Processes**. The instances of these component classes are assembled at runtime to a network of components that connect their input and output interfaces by creating object bindings, that update each other by sending **messages** through a **Binding Service**. The application can be modified and optimized at runtime by rearranging the components in the network. The application consists of objects that perform tasks in **steps** that can be grouped into **transactions**. The completion of transactions and the **state** of objects can be tracked by a **Transaction Management Service** that initiates re-execution in the case of failure or the need of relocation (e.g. idle CPU cycles no longer available). Global data storage can be enhanced redundant and distributed, by configuring mirrored objects that are synchronized by a **Message Mirroring Service**.

Cluster Server Processes are assigned to one or several **Object Class Domains**. Object Class Domains are configurable sets of cluster server processes where instances of individual **Python** classes can be **(re-) located** to. This is an object distribution configuration mechanism, to control the computation and network **load distribution** appropriately for the network and application structure. This way the available **resources** can be classified so that they are used in an optimal way.