High Performance Distributed Object Systems

Dennis Gannon

Department of Computer Science, Indiana University, Bloomington, IN 47401

Abstract. This paper will provide a survey of current work on object oriented tools and techniques for metacomputing systems. More specifically, we consider the problem of designing a software component architecture that extends the current emerging desktop object composition models to the domain of high performance networks and massively parallel compute servers.

1 Introduction

Successful software development must now live by a new set of rules: A drive toward more end-user programmability. A movement away from monolithic applications and toward systems grown from the integration of many smaller components. Increased expectations of radically greater functionality, delivered under radically shorter schedules. Grady Booch [2]

The design of the current generation of desktop software technology differs from past generations in one fundamental way. The new design paradigm states that applications should be built by composing “off the shelf” components, much the same way that hardware designers build systems from integrated circuits. Furthermore, these components may be distributed across a wide area network of compute and data servers. Each component is defined by the public interface that specifies its function as well as the protocols with which other components may communicate with it. An application program in this model becomes a dynamic network of communicating components. This basic distributed object design philosophy is having a profound impact on all aspects of information processing technology. We are already seeing a shift in the software industry toward investment in software components and away from hand-crafted, stand-alone applications and within the industry a technology war is being waged over the design of the component composition architecture.

High performance computing will not be immune from this paradigm shift. More specifically, as our current and future Internet continues to scale in both size and bandwidth, it is not unrealistic to think about applications that might incorporate 10,000 active components that are distributed over that many compute hosts. Furthermore, pressure from the desktop software industry will compel us to integrate the applications that run on supercomputer systems into distributed problem solving environments that use object technology. Metacomputing systems consisting of MPP servers, advanced, networked instruments,
database servers and gigabit networks will require a robust and scalable object model that support high performance application design.

This paper will sketch the current software design ideas that make up the commercial desktop software environment. We will then explore where these ideas scale well and are applicable to scientific and engineering applications and where they fail to meet the needs of high performance computing. We conclude with a brief survey of several approaches to high performance object systems now in use in research settings.

2 Basic Concepts and Examples

The basic ideas behind Object Oriented software design that are important for this paper are as follows.

- Data, and the functions that operate on it, should be bound together into objects. These objects are each instances of an abstract data type called a class. The data associated with an object are called data members or attributes and the functions that are associated with a class of objects are called member functions.

- Interfaces describe a set of functions that can be used to interact with a family of objects. Those classes of objects that respond to a particular interface are said to implement that interface. A class of objects may implement more than one interface.

- A new class of objects may be built from an existing class by adding new data attributes or member functions. Instances of the new class each contain an instance of the original (parent) class and thus, still implement the same interfaces as the parent. This process of extending one class to build another is called inheritance: the extended class is said to inherit from the original class. It is also possible for the extended class to override the definition of a member function from a class that it extends. Consequently, it is possible to make an extended class specialize or modify the behavior of the parent, i.e. it responds to the same functions, but the action is different.

- It is also possible to create a new interface definition by extending the definition of one or more other interfaces by simply adding new functions.

Consider the following example. Suppose we are designing a new aircraft and we wish to simulate the aerodynamics of the new vehicle. The components of the design application might include a CAD design database which represents the new structure in a special polygon based format. The aerodynamic simulation will require a flow solver that requires a 3-D grid of the exterior of the vehicle as input and generate a flow field as an output. Another component would transform the CAD database description into the grid structure. A visualization system could be used to display the flow field. The interface to the CAD database might look as follows. It has a single member function that takes the name of the design and returns a stream of polygons. To describe this we can use an Interface Description Language (IDL) which might take the form