Building Blocks for Client and Server Functionality in Group Protocols: The Funclets Pattern

Bela Ban
Dept. of Computer Science
Cornell University
bba@cs.cornell.edu

Sep 4 1998

Abstract

Typical protocol implementations consist of message exchanges between communicating entities. These entities often assume either a server or a client role, depending on the initiator of the communication. The roles may vary dynamically, for example, in group communication protocols, a coordinator entity may fail, and a different entity has to take over its role, thereby changing from server to client (initiator).

Frequently the communication patterns and algorithms used in a specific domain such as group commuication protocols are well known and used over and over again. The goal of this paper is to show that they can be captured into *classes of algorithms* and subsequently be reused by protocol implementors, thereby reducing time for developing and testing new protocols.

A challenge of providing reusable classes of algorithms is that they should be usable both in the client and server roles. At first sight, this may be solvable by parameterizing instances of algorithms with their roles (e.g. client or server), and - depending on their role - invoke different code. However, the bigger challenge is how to *integrate* these algorithms with existing code. It is for example easy to provide an algorithm in class form that can be called by a client. However, it is significantly harder to make an existing server call this functionality, without actually copying and pasting all methods of the class into itself (thereby destroying encapsulation and reusability).

This paper presents a mechanism that allows to add prepackaged functionality to entities in client or server roles without major code modification. As an example, we present a *state transfer protocol* building block which - when added to client and server - allows new group members to aquire their initial state from an existing group member.

1 Motivation

When implementing protocols for group communication or applications based on group communication one frequently encounters the same (or similar) patterns of communication. For example, a distributed group commit protocol would typically have a coordinator send a multicast message with the proposed modification to all group members. Each member records the message on stable storage and returns an acknowledgement. When all members have acknowledged, the coordinator sends a commit message, causing all members to deliver the message to the application. In the case of a member crash, the coordinator sends an abort message to the group members.

This communication pattern lends itself to be packaged into a reusable *algorithm class* (*funclet*) to be used by clients (coordinator) and servers (group members). The above sequence of message exchanges would not have to be written from scratch, but a tested and tailorable algorithm could be used instead, resulting in more secure, smaller and better maintainable code. Because an algorithm is captured in one single place (a class) with a fixed public interface, it is easy to improve the algorithm's implementation without causing any changes in its users. Applications or protocols don't have to be straddled with pieces of the logic of the algorithm, but the latter is contained in a single class, to which they can delegate their execution.

As shown in the example, algorithms may be *asymmetric*, that is, the communicating entities either act in the client role or in the server role.[1] Depending on the role of their hosts, funclets are accessed differently: in the client role, a client simply invokes a method of a funclet instance, causing a communication exchange to be started. In the server role, the server typically wishes to *delegate* the performer role of the service to a funclet which acts on behalf of the server for a certain *range of incoming requests*, for which the funclet registered with the server when it was created.[2] A building block underneath the server determines whether there are any funclets registered and, if so, checks whether the request is intended for the funclet. If this is the case, the request will be entirely handled by the funclet, otherwise (by default) the server will receive the request. This scheme allows to add functionality to a server entity without actually having to change its implementation.

[1] A client is the *initiator of a communication*, and a server the *performer of a service*. During a communication exchange, these roles may change dynamically, e.g. a server may become a client by requesting a service from a different entity.

[2] How the range of requests is defined exactly is discussed later.