Bond system monitor

Kyung-Koo Jun, Ladislau Bölöni, Ruibing Hao, and Dan C. Marinescu
(junkk, boloni, hao, dcm@cs.purdue.edu)
Computer Sciences Department, Purdue University
W. Lafayette, IN 47907, U.S.A.

August 6, 1998

Abstract

Bond is a message-oriented middleware for network computing on a grid of autonomous nodes. It consists of a distributed object communication fabric, servers, and agents. Core servers are permanent objects providing services such as directory service, dispatching, authentication, monitoring, and others. The Bond system monitor is responsible for starting up core servers, keeping them running, and balancing the load among them. In this paper, we discuss the remote server start-up and the executables failure detection functions of the monitor and present the backup monitor.

Contents:

1. Abbreviations and terms

Backup system monitor: core server; it takes control of the system when the primary system monitor fails.
Shadow of a Bond object: abstraction supporting communication with a remote object. A shadow is a local object acting as a proxy for the remote object, like CORBA's stubs.
DSVN – Directory Server Virtual Network: collection of shadows of directory servers. This object allows any server to communicate with the directory servers.
Workspace: a container object. It provides references to a subset of local objects.
Monitor-info object: an object that provides information to the system monitor. This object is created at the workspace creation time.
Sub-protocol: sub-set of KQML messages used by objects to implement different functions. It is equivalent to method invocation in CORBA or RMI.
Performative: component of a KQML message describing the action to be performed.

Primary copy: the original object replicated to a remote workspace.
Primary system monitor: core server responsible for configuration management, availability and load balancing.
ResidentVN - Resident Virtual Network: a collection of shadows of residents.
Secondary copy: the replicated object in a workspace from the primary copy.
Subscription: the action allowing a workspace called subscriber to monitor another workspace called publisher.
SystemVN – System Virtual Network: a collection of shadows of all active executables.

2. Introduction

Bond is a Message Oriented Middleware, MOM, for network computing on a grid consisting of autonomous nodes. It consists of a distributed object communication fabric, servers, and agents. Core servers are permanent objects providing services such as directory service, dispatching, authentication, monitoring, and others. Agents provide support for activities like scheduling, brokerage, remote execution of computations, and so on. The survey of Bond is presented in [1],[2],[3],[4] and reviewed briefly below.

The Bond objects are persistent network objects, communicate with each other, can be instantiated and run remotely, and can be saved on permanent storage [2],[4]. Objects in the memory are said to be *active* while objects in the secondary storage are *passive* [3].

The Bond object hierarchy is presented in [3]. In this hierarchy:

bondObject: is the root of the hierarchy. It implements the common fields of all Bond objects (e.g., name, unique *bondID*, address, and type).
bondData: is the common ancestor for the objects that represents persistent data in the external storage.
bondExecutable: is the common ancestor of the classes that implement executable programs such as servers and agents

Shadow objects provide a high level abstraction for a unidirectional communication channel linking two objects together [1]. When an object needs to communicate, it involves a *directory service*. If the object is found, a shadow of the remote object is created and the connection is established. The *say* method of a shadow object allows a string to be transmitted to the remote object whose *say* method in turn is invoked to process the message. The *realize* method creates a local copy of the remote object by serialization [3].

A *virtual network* is an abstraction for a set of objects functionally related to one another [1],[4]. It is a local collection of shadow objects. An object creates a virtual network to manage a set of related remote objects. For example, a directory server maintains the virtual network of the registered executables [3].

KQML, the *Knowledge Query and Manipulation Language* [5],[6] is used as an inter-object communication language. KQML messages, called *performatives*, allow to encode basic abstractions like asking, replying, achieving, subscribing, or notifying while the contents of the messages are partially encoded in parameters and partially assuming to be known by both parties [2]. There are several classes of performatives: informative performatives like *tell* and *deny*, effector performatives like *achieve* and *unachieve*, notification performatives like *subscribe* and *discard*, networking performatives like *register* and *unregister*.

Sub-protocols [2] are sub-sets of the KQML messages, exchanged among objects in a particular semantic relationship like client-server interactions, monitoring, event handling, cooperation, etc. For example, the property access sub-protocol is used for retrieving and modifying properties of objects.

The basic element of Bond architecture is a *cell*, a collection of Bond objects, coexisting on a given host [1]. The cell consists of a *local directory* (a collection of references for local objects), *a resident* (the main control thread of the cell), other threads spawned by the resident, including a *messaging thread*, and two *mailboxes*, an *inbox* and an *outbox*.

The local messaging thread waits for messages delivered to its *inbox*, then parses them to determine the local destination objects and invokes the *say* method on the target object. The *say* method of the shadow places a message in the *outbox* of the messaging thread, which in turn transmits it to the *inbox* of the destination cell.

A *workspace* is a container object for collection of references to local objects. In case of a remote object, the local copy should be created by the *realize* method of the shadow. A particular instance of the workspace is a user’s workspace consisting of all objects owned by the user. A *Bond executable is* an ancestor of all executable programs and has one workspace. The executable is called the *owner* of the workspace. The workspace contains the virtual network of the owner.

The descendents of the Bond executable object are *servers*, *agents*, *residents*, and *wrappers* [3]. Servers are permanent; their life-time is of the same order as the life-time of the system and provide system-wide services like *directory service*, *authentication & software distribution service*, *monitoring*, *persistent storage service, dispatching*. Agents are started upon request e.g., *scheduling agents* and *security agents*. They work in conjunction with one or more objects and terminate at the completion [4]. *Residents* are the main threads of control of a cell and Bond executables are started from them, as discussed later. The functions of *wrappers* are discussed in [3].

3. System monitor

The *system monitor* is a server with three functions: (a) configuration management, remote start-up and shutdown of servers, (b) error and failure detection and recovery, and (c) core servers load balancing.

The system monitor has three virtual networks*: the System Virtual Network (SystemVN), the Resident Virtual Network (ResidentVN)*, and the *Directory Server Virtual Network (DSVN)*. The SystemVN consists of the shadows of all active executables. The ResidentVN and the DSVN contain the shadows of residents and directory servers respectively. In this section, we discuss the server start-up, the failure detection, and the *backup monitor*.

Remote server start-up

Bond executables run within a cell. A cell must be started on a host to run an executable. Cells are started from the command line using an *initiator* or by remote execution methods provided by the operating systems. The initiator initializes the cell, and leaves the cell running with two threads, a resident and a messaging thread [1]. Other threads running Bond executables are started remotely by the system monitor or locally from the command line parameter of the initiator.

The system monitor uses the *agent control sub-protocol* to start up the server. Once started the server must register with both the system monitor and one of the directory servers using the *registration sub-protocol*. At least one directory server should be available if a cell exists. The system monitor starts a directory server on the first registering cell automatically.

The agent control subprotocol is used to control the execution of the Bond executables. It consists of two performatives as shown in Table 1. The registration subprotocol is used by the Bond executables to register with the system monitor and the directory server. It consists of two performatives as shown in Table 2.

Performative	:content	Parameters	Description
achieve	start-agent	:agent *type of object*	ask to initiate the execution of the object
tell	new-agent	:agent *ID of new object*	notify the start-up

Table 1. Agent control sub-protocol

Performative	:content	Parameters	Description
achieve	register	:name *bondID* :address *address*	A Bond executable named *bondID* at *address* registers with the system monitor and the directory server
recommend-one	*type*	:name *bondID* :address *address*	The system monitor recommends an executable the *type* server named *bondID* at *address*

Table 2. Registration sub-protocol

Figure 1 illustrates the server start-up process on a cell: the system monitor sends to a resident a request to start up a server; the resident starts up the server, which in turn registers with the monitor; the system monitor recommends one directory server with which the server should register. The system monitor is started from the command line parameter of the initiator.

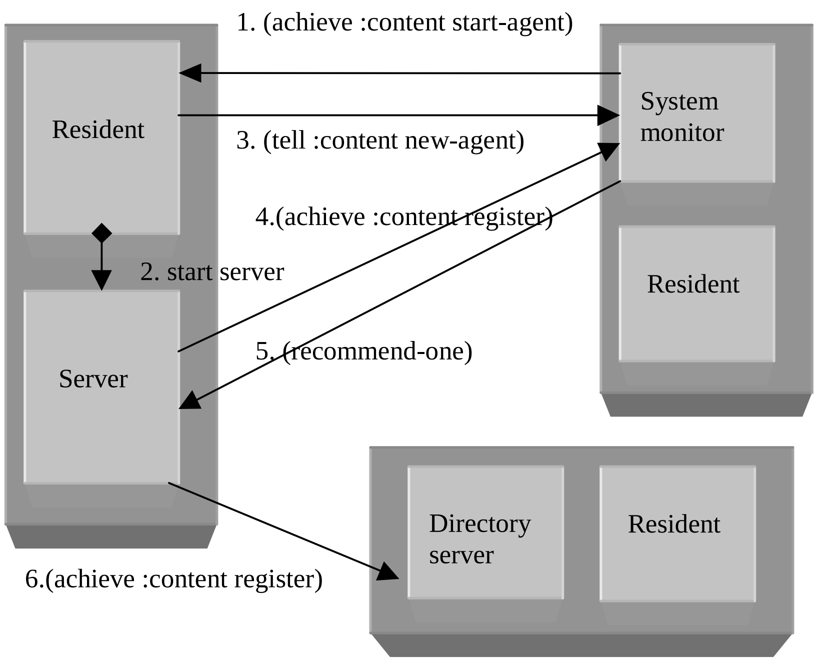

Figure 1. The server start up procedure on a remote host. The resident running on that host is contacted by the system monitor using the agent control sub-protocol. Once started the server registers with the monitor and the directory server using the registration sub-protocol

Failure detection

The system monitor is responsible for detecting the failure of servers and residents. We first discuss *workspace synchronization.*

An executable has one workspace containing references to local objects. *Synchronization of workspace* is an abstraction for replicating active objects in a workspace. The original object is called a *primary copy* and the replicated object is called a *secondary copy*. An executable may subscribe to a set of objects in the workspace owned by another executable. In this case, the secondary copy of the object subject to synchronization is guaranteed “to follow” the primary copy.

All properties of the primary copy are replicated. The subscription relationship is established between two workspaces using the *monitoring sub-protocol*. The workspace that contains primary copies is called a *publisher* and the one that includes secondary copies is called a *subscriber*.

The *monitoring sub-protocol* is used by publishers and subscribers for workspace synchronization. In addition, it allows one executable to monitor other executables. The sub-protocol consists of four performatives as shown in Table 3. The *subscribe* and *discard* performatives are used to initiate and terminate a subscription. The *subscribe* performative has two parameters: *interest* specifies objects by Bond ID or type; *frequency* specifies the desired message interval.

Publishers send periodically, at specified intervals, *tell* performatives to subscribers. The bondID and address of the primary copy, included as the parameter *value* of the message are used to construct the shadow, then the *realize* method is invoked to create the secondary copy. The parameter *modified* (*yes/no)* indicates whether the primary copy was changed since the last message. If a message does not arrive within the expected interval, the subscriber notifies the publisher of the missing message, using the *ask-one* performative.

Performative	:content	Parameters	Description
subscribe	monitor-info	:interest *messages* :frequency *interval*	ask to send *interest* at an *interval*
discard	monitor-stop		ask to stop sending
ask-one	monitor-poll		ask immediate reporting
tell	monitor-report	:value *messages* :modified *yes,no*	reply with the requested data

Table 3. Monitoring sub-protocol

The system monitor uses the workspace synchronization to detect the failure of servers and residents. The monitoring shown in Figure 2 consists of the following steps:

1) The system monitor subscribes to servers and residents.
2) The executable being monitored replies with periodic messages.
3) The system monitor checks the message interval by the timer.
4) If a periodic message is not received, the monitor times out and sends inquiry messages.
5) No response to the inquiry messages is considered as the failure of the executable.

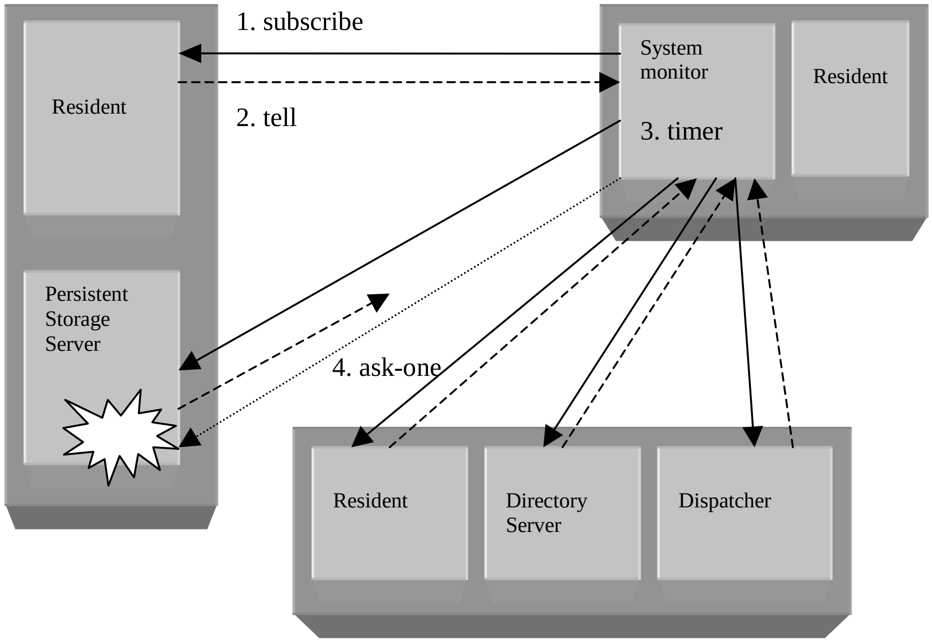

Figure 2. The system monitor subscribes to all executables and receives the periodic messages. The time-out triggers the monitor to send the inquiry message.

We discuss in more detail the mechanisms used for monitoring. The system monitor subscribes to the *monitor-info* object of the executables using the *subscribe* performative. The monitor-info is the object containing the bondID and the shadow of the executable. Every executable has one monitor-info, created by default when its workspace is instantiated.

The *monitoring-message* sent periodically by the monitored executable consists of the *tell* performative and the bondID and the address of its monitor-info object. The system monitor constructs the shadow of the executable's monitor-info object and creates a local copy, using the *realize* method. Then it adds to this object a new property, *due-time,* that specifies the time by which the next monitoring-message should arrive. Whenever monitoring-messages arrive, a new due-time is calculated considering both the current time and the subscribed interval.

The system monitor expects periodic monitoring-messages from every executable it has subscribed to and decides that an executable is not running if a monitoring-message is overdue. This is a *fail-stop failure* [7]. To confirm the failure, the system monitor sends an *ask-one* performative using the bondID and the shadow of the executable available from the *owner and owner-shadow* properties of the monitor-info object. The number of inquiry messages is recorded in the *How-may-ask-one* property of the monitor-info object. If this number exceeds a pre-defined limit, the system monitor concludes that the executable is down.

Backup monitor

To tolerate the failure of the primary system monitor a backup monitor is provided. The primary monitor starts up and monitors the execution of Bond servers including the backup monitor. The backup monitor runs on standby at first and replaces the failed primary monitor.

The primary monitor is started by a system administrator, then it starts up the backup monitor on a remote cell. The backup monitor subscribes to the SystemVN of the primary that in turn subscribes to the monitor-info object of the backup monitor. The primary and the backup monitors start exchanging periodic messages.

The backup monitor not only detects the failure of the primary but also replicates the SystemVN of the primary. The DSVN and ResidentVN of the backup are constructed and updated from the local copy of the SystemVN.

In case of the failure of the backup monitor, the primary just initiates another backup monitor. Figure 3 illustrates the primary monitor failure. The backup monitor becomes a new primary and subscribes to monitor-info objects of all active executables in its SystemVN. The subscription lets the executables replace the primary monitor shadow with the backup monitor shadow in their virtual networks. A new backup monitor is started.

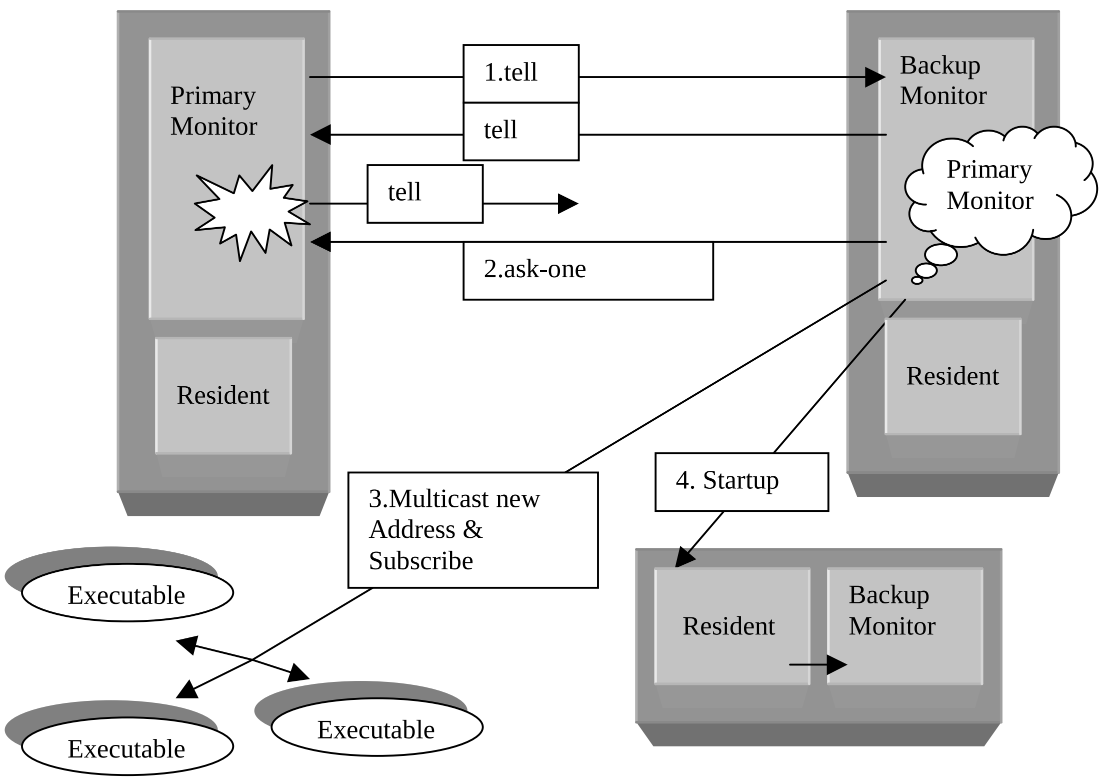

Figure 3. Once the backup monitor discovers the primary monitor is down, it becomes the primary monitor. It multicasts its address and subscribe messages to active executables. Another backup monitor is started up.

4. Future work

Future work consists of a procedure for remote server shutdown, synchronization models, and the directory servers load balancing.

The remote server shutdown procedure should be modified to allow self-termination. The virtual networks containing the shadow of the terminated server should be notified about the change; the servers should identify a list of local objects, which must be saved before termination and restored later e.g., a virtual network of servers.

Two new synchronization models will be added. In the *Immediate synchronization model* property changes of the primary copy trigger the "immediate" notification of the subscribers. *Smart synchronization* is the hybrid of delayed and immediate models. The publisher notifies the changes immediately and it sends a periodic message if there is no change during the subscribed interval. Each subscription is named after the synchronization models. The *delayed subscription* is proper for the failure detection, the *immediate subscription* for event handling, and the *smart subscription* for consistency between primary and secondary copies.

Since directory servers periodically poll the status of registered executables, the directory server with a large number of registrations may be overloaded. The system monitor should distribute executables evenly to each directory server. The system monitor subscribes to directory servers to obtain the number of registered executables and recommends the directory server with the smallest number of registered executables.

5. Conclusions

In this paper we present the system monitor, its functions and the backup monitor. The system monitor is the core server providing the server start-up and the monitoring service based on the workspace synchronization. We discuss the agent control sub-protocol used by the system monitor to control executable objects, the registration sub-protocol used by an executable object to register with the system monitor and a directory server, and the monitoring sub-protocol for workspace synchronization. The workspace synchronization is an abstraction for replicating objects between workspaces. The delayed synchronization occurs at the fixed intervals.

References

1. L. Boloni, K. Jun, M. Sirbu, and D. C. Marinescu. *Seamless Metacomputing with Bond.* Technical Report CSD-TR #98-010, Purdue University, April 1998.
2. L. Boloni, K. Jun, T. Daniels, and D. C. Marinescu. *Message patterns in the Bond Distributed Object System*. Technical Report CSD-TR #98-004, Purdue University, March 1998.
3. L. Boloni, *Bond Objects –- a white paper*. Technical Report CSD-TR #98-002, Purdue University, February 1998.

4. R. Hao, L. Boloni, and D. C. Marinescu. *Bond System Security and Access Control Models*. Technical Report CSD-TR #98-019, Purdue University, June 1998.
5. T. Finin, et al. *Specification of the KQML Agent-Communication Language*. DARPA Knowledge Sharing Initiative draft, June 1993.
6. Y. Labrou and T. Finin. *A Proposal for a new KQML Specification*. UMBC TR-CS-97-03.
7. P. Stelling, I. Foster, C. Kesselman, C. Lee, and G. Laszewski. *A Fault Detection Service for Wide Area Distributed Computations*. Proc. 7th IEEE Symp. on High Performance Distributed Computing, to appear, 1998.