UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

DEADLOCK AVOIDANCE IN DISTRIBUTED SERVICE ORIENTED ARCHITECTURES

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

JOHN MATTHEW MARTIN

Norman, Oklahoma
2010

DEADLOCK AVOIDANCE IN DISTRIBUTED SERVICE ORIENTED ARCHITECTURES

A THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY

Acknowledgments

First, I would like to thank God for His many blessings and for giving me the opportunity to write this thesis. Next, I would like to thank my wife, Jaime Martin, and my parents, John and Elaine Martin, for their continual support and the many sacrifices they have made that have enabled me to be in the position that I am today.

I would also like to thank my advisor, Dr. John K Antonio, for his guidance and direction throughout the course of my graduate studies. In addition, I would like to thank the members of my thesis committee, Dr. Sudarshan Dhall and Dr. Dean Hougen. I appreciate their questions, comments, and advice.

Finally, I would like to thank the following members of our research team for their contributions to this thesis: Kelly Crawford, Nicolas Grounds, Jason Madden, Jeff Muehring, Jay Sachs, Carlos Sanchez, Tom Stockdale, Swathikah Thangaraja, and Josh Zuech. I'm very grateful for the insight and advice provided through various discussions and meetings with the research team.

Contents

List of Tables

List of Figures

Abstract

A distributed service-oriented architecture comprises interconnected machines that together support a number of computational services. Concurrent service requests made to an individual machine are supported with shared, and limited, resources associated with that machine. A call to a service method may in turn invoke methods from other services, resulting in a nesting of service calls that is represented by a call tree. Deadlock occurs when a circular dependence is formed as a result of requests (calls) waiting for machine resources to be released by other requests. A deadlock avoidance technique is derived from Dijkstra's Banker's Algorithm that accepts or denies preferred scheduling and method-to-machine assignments proposed by underlying policies. Assumed to be known and available are estimates for the resource requirements of methods and the structures of the call trees. Simulation studies are conducted that demonstrate the effectiveness of the approach in avoiding deadlock, while not degrading (and in most cases improving) the performance of the underlying policies.

Chapter 1

Introduction

As the demand for computing power is ever increasing, so is the use of parallel computing and parallel systems to meet this demand. Traditionally, software has been written for serial computation; that is, to be run on a single computer having a single central processing unit or CPU. In this paradigm, a program is broken into a series of discrete instructions to be executed one after another, only one of which may execute at a given moment in time. Conversely, in parallel computing, a problem is broken into smaller subproblems that can be solved concurrently. Each subproblem is then broken down further into a series of instructions that can be executed simultaneously on different CPUs. Solving problems in this parallel and distributed fashion can lead to a significant reduction in the amount of time required to solve a given problem [1].

Historically, parallel computing has been used mostly for solving complex scientific and engineering problems, especially when massive amounts of data must be processed. Some classic examples employing the use of parallel computing include applied physics and mathematics, biomedical analysis, speech recognition, and weather forecasting [2]. As time has progressed, the use of parallel computing and parallel systems has become more common. Some more recent applications of parallel systems include financial modeling and forecasting, web search engines, large social networking web sites, and large e-commerce web applications.

1.1 Benefits of Parallel Computation

Parallel computation is used primarily because of the speedup it provides. It comes as no surprise that parallel systems have the potential for significant speedup because a large problem can often be attacked from multiple angles concurrently. A good example of such a problem is one involving the processing of large data sets or requiring many iterations or simulations to reach a solution. For these types of problems where the running time of a sequential solution may be simply unacceptable, parallel systems provide the only feasible solution [2]. In addition to providing speedup, parallel systems can also increase the precision of a particular solution [3]. Consider the case in which a solution to a hard problem can be reached in a specified amount of time using a sequential algorithm, but only if some simplifying approximations are made. If the same problem was to be solved in parallel, the assumptions may no longer be needed as more CPU resources could be devoted to the subproblems. As more resources are devoted to solving the problem, the precision of the solution can be increased.

For a real world example of a hard problem in which parallel computing is used to obtain a solution, consider the problem of searching the internet via www.google.com. When a user enters search text into the browser, it is paramount that the results return to the user as fast as possible. For a problem of this size, it is simply infeasible to search the entire space with a sequential algorithm using only a single CPU. For this reason, Google uses parallel and distributed systems to perform the search [4].

1.2 Parallel Algorithms

As a general rule, computers that are manufactured today consist of more than one CPU. As of the date of this writing, it is not uncommon for a commodity workstation to have 4 or even 8 CPUs, with high performance computers having many more. These multi-core machines can be linked together to provide massive computational resources called distributed systems [5]. This means that it falls to software engineers and application developers to design and implement progamming models and algorithms that take advantage of the increasing amount of parallelism available. This is no easy task and is an ongoing challenge facing software developers today.

For an example of a programming model that capitalizes on the amount of available parallelism, consider MapReduce [4]. MapReduce is highly scalable and flexible, allowing users to take advantage of parallel and distributed systems without prior experience in the area. It was inspired by the map and reduce primitives found in Lisp and many other functional programming languages. In this model, the user writes two basic functions: a `Map` function and a `Reduce` function. The `Map` function takes as input a key/value input pair and produces a set of intermediate key/value pairs. The `Reduce` function then accepts an intermediate key and a set of values for that key and merges them together into a set of output values [4].

For an illustrative example of MapReduce, consider Fig 1.1. First, the `Map` function considers the set of inputs and distributes them to the various compute nodes as indicated by the arrows in the figure. The compute nodes then perform the calculations to produce intermediary outputs that are then collected and reduced by the `Reduce` function to generate the set of outputs. For this model to work, it must be the case that the original problem can be divided into a set of subproblems that the compute nodes can work on in parallel. For a more detailed overview and description of MapReduce, see [4].

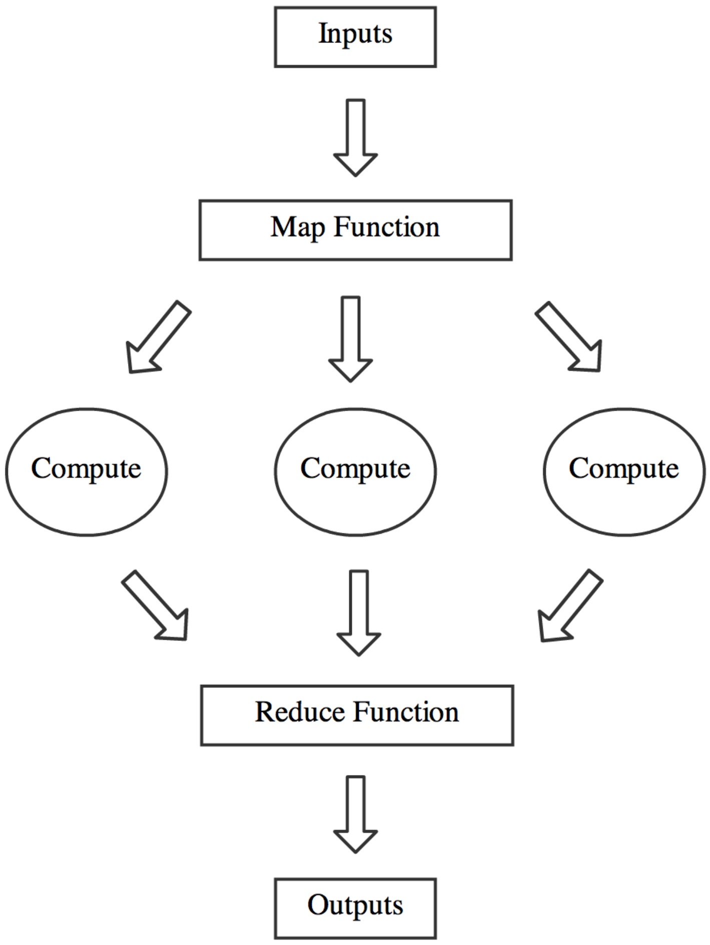


Figure 1.1: An overview of the MapReduce programming model.

1.3 Service Oriented Architecture (SOA)

This thesis deals with the problem of scheduling computations in a type of parallel system called a Service Oriented Architecture or SOA. An SOA is a way of organizing software components to increase overall system efficiency and performance. More specifically, an SOA provides a flexible architecture by modularizing large applications into services that communicate with each other through specified service interfaces using service requests. SOA is more than just a way of organizing software. It is a way of designing a software system to create a design style, technology, and framework that all facilitate a cost-effective and logical way of developing and deploying software [6]. A major advantage of the SOA concept is that the architecture allows software developers to take advantage of the power of parallel computing without having to necessarily understand every part of the larger system and how the parts inter-operate amongst themselves.

A service in an SOA is a well-defined, self-contained software module that implements some business functionality and is independent of the state or context of other services [6]. Services are defined using a standard definition language, have a published interface, and communicate with each other to collectively support a common task or business goal. A service within an SOA must be autonomous. That is, the service operations must be opaque to other external components of the system. The only information external components know about a particular service is how to communicate with it through its published interface. This provides for the loose coupling among services which is a hallmark of an SOA. Because of this structure, an SOA provides a way to reduce the overall system complexity by encapsulation. Each service encapsulates some part of the overall system logic. This makes the overall system easier to manage and deploy.

An SOA is made up of two main participants: a service consumer (requester) and a service provider. These participants communicate with one another using a predefined protocol via service requests and service responses, as illustrated in Figure 1.2. A service request is a message adhering to a known standard such as the Simple Object Access Protocol [7] or SOAP used in Web services. SOAP is a standard protocol for exchanging structured information over the Internet and for making service calls to a Web service. A SOAP message includes a header and a payload section. The header contains the information needed to properly route the message to an endpoint and the payload is the data on which the endpoint operates. An SOA does not necessarily imply the usage of SOAP, but SOAP is a very common message format found in SOAs.

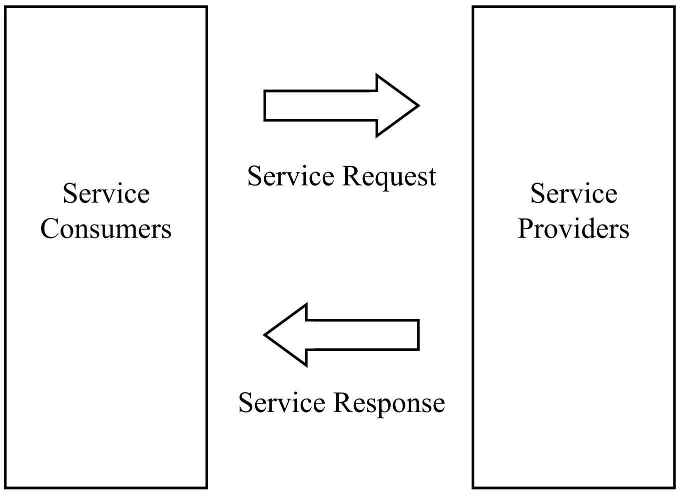


Figure 1.2: Communication between service consumers and service providers.

A service requester can use SOAP or some other mechanism to directly share information with the service provider, but there is also another role in an SOA that makes information sharing more robust. That role is called service aggregation and fits the description of both a service requester and a service provider [6]. The main job of the service aggregator is to provide one location that houses all of the services. This alleviates some of the difficulties with discovering and locating new services coming online as the functionality of the SOA evolves. In addition, a service aggregator may contain additional metadata about the services it houses and can optionally make that data available to all service clients.

In addition to the concept of service aggregation is the concept of service orchestration. Service orchestration is the process of integration, coordination, automation, and management of services in a composite service environment [5]. Software that facilitates communication between web services is called middleware. Some well known, established middleware technologies include JMS [8, 9], CORBA [8, 9], AMQP [10], and Open-MQ. These technologies provide a layer where communication and coordination occurs between the services. This layer is sometimes called an Enterprise Service Bus [6] or ESB. The ESB is responsible for providing a layer for

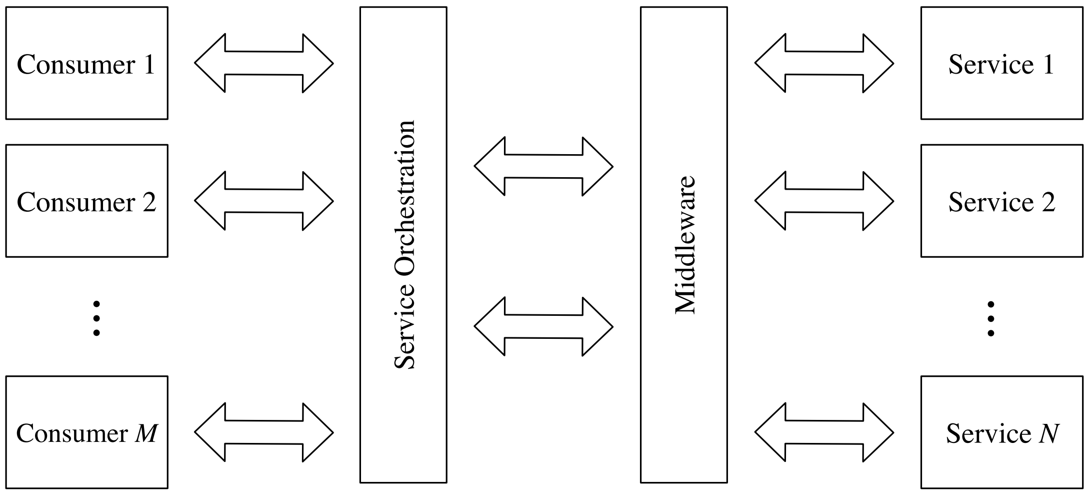


Figure 1.3: Service orchestration in an SOA.

the implementation, deployment, and management of an SOA. Fig. 1.3 illustrates how service orchestration can be accomplished using middleware.

1.4 The BlueBox SOA

Work started on BlueBox in 2003 at RiskMetrics Group as part of an effort to provide a highly scalable software architecture to address the growing needs of clients. Since then, BlueBox has developed into a general-purpose, distributed SOA with inherent support for non-domain-specific utilities and features such as AAA authentication (authentication, authorization, and access control) and process monitoring and tracking. The architecture of BlueBox is illustrated at a high level in Fig. 1.4. Clients interact with BlueBox via the Internet through a firewall. The topmost layer of the BlueBox system shown in the figure illustrates several different gateways through which clients interact with BlueBox, including web applications, ftp sites, and webservices. The layer just below the client facing layer is a messaging layer through which all communication occurs in BlueBox. In this layer is the Enterprise Service Bus (ESB), implemented in Java Message Service (JMS). Applications and services use JMS to communicate with each other through JMS messages. Finally, the bottommost layer in the figure represents the Framework layer. The Framework is a service aggregator [6] that is responsible for loading and unloading the services that make up BlueBox. Services typically have multiple instances and may depend upon resources outside of the BlueBox environment. This is represented by the arrow pointing out from Service 1 to an external cloud and the arrow pointing out from Service n to an external database. In a typical BlueBox environment, multiple Framework's are deployed, each loading an arbitrary number of services.

1.5 Problem Addressed in Thesis: Deadlock in an SOA

In a distributed SOA [6], services define the different categories of computational operations that are available. Each service includes a number of associated service methods. Each machine in an SOA contains one or more service instances and must supply the resources necessary to carry out the methods associated with those instances. In general, machine resources include available memory, CPU capacity, persistent storage, I/O resources, network resources, and threads. The primary machine resources considered in the simulations conducted in this thesis are memory and CPU.

The implementation of a given method may itself call upon other methods, and the called methods are not necessarily executed on the same machine as the calling method. Furthermore, method calls can be nested to an arbitrary depth; nested method calls are represented by a call tree. Fig. 1.5(a) illustrates two simple call trees. The root node of a call tree corresponds to the method that defines the entire scope of the call tree.

Fig. 1.5(b) illustrates the order in which the method segments of the calling

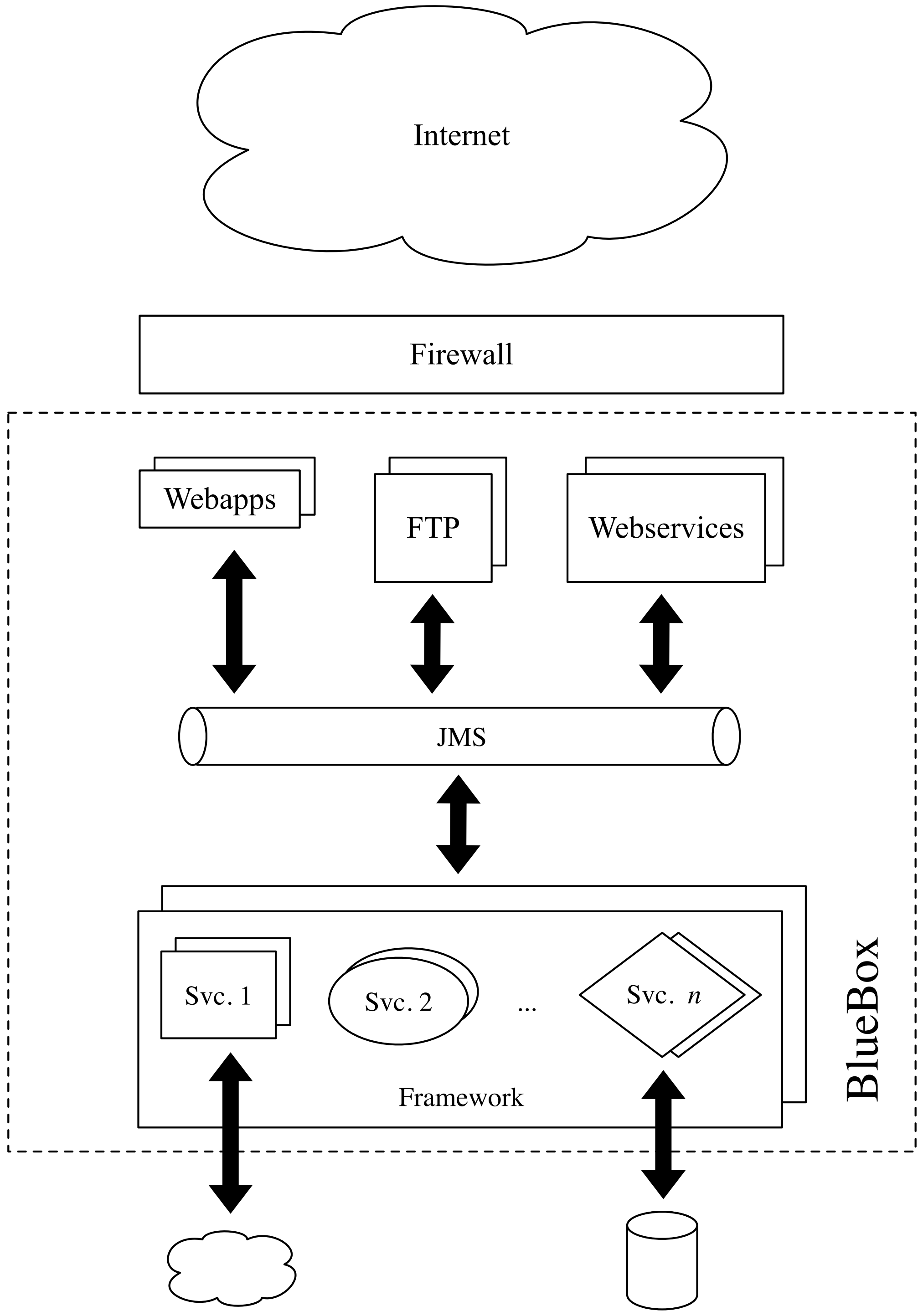
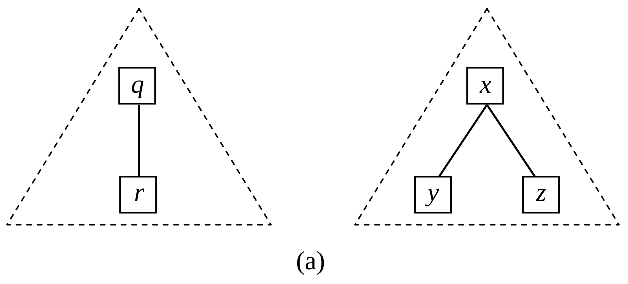


Figure 1.4: BlueBox architecture.



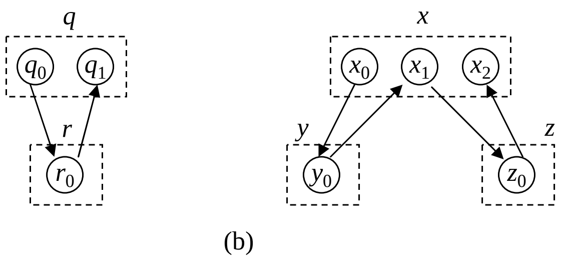


Figure 1.5: (a) Example of two call trees. (b) Illustration of the method segments of the call trees shown in (a).

methods of the two call trees of Fig. 1.5(a) are executed. The boundary between adjacent segments of a method is defined by the point in the calling method where a call is made. For example, method q first executes segment q_0; after q_0 executes to completion, method q then calls out to method r. Upon r's completion, q resumes execution at segment q_1. Leaf methods, by definition, do not call other methods, and thus the entirety of a leaf method is defined to be a single segment.

During the time that a called method (e.g., method r in Fig. 1.5) is executing, the calling method (e.g., q) is defined to be in a holding state. For the purposes of this thesis, a method in a holding state does not consume CPU resources, but it does consume (i.e., hold) memory resources of the underlying machine to which it is assigned. A method that is executing (defined when one of its segments is executing) generally consumes both CPU and memory resources of the machine to which it is assigned. During the execution of a call tree, at most one method of the call tree is actively executing at a time; however, one or more methods may be in the holding

Table 1.1: Machine assignments and memory requirements for service methods of call trees in Fig. 1.5.

Service Method	Machine Assignment	Memory Requirement
q	machine 1	0.6
r	machine 2	0.4
x	machine 2	0.7
y	machine 1	0.6
z	machine 1	0.3

state.

Concurrent execution of two or more call trees in a distributed SOA can lead to deadlock. To illustrate how deadlock can occur, consider the execution of the two call trees of Fig. 1.5 on an SOA that contains two machines. Further assume that the service methods of the call trees are assigned to machines according to Table 1.1. This assignment of methods to machines is illustrated graphically in Fig. 1.6. In addition to machine assignments, Table 1.1 also defines the memory requirement of each method; it is assumed that each machine has a normalized memory capacity of unity that cannot be overcommitted. For this scenario, deadlock may or may not occur depending upon how the execution of each call tree's segments are scheduled.

As illustrated in Fig. 1.7, deadlock does occur if the execution of q_0 and x_0 overlap in time. In the figure, time progresses in the downward direction, and the shading of vertical bars indicate that the corresponding segment is executing. After completion of x_0, x calls out to method y, which is assigned to execute on machine 1. However, y_0 cannot be granted permission to begin executing on machine 1 because of insufficient memory capacity; the total memory requirement of q and y is 1.2, which is beyond machine 1's assumed capacity of unity. Similarly, upon q_0's completion, it calls out to r on machine 2, but r cannot begin execution because the total memory requirement of x and r is 1.1. As a result, q and x are deadlocked.

Recognizing that the execution of q_0 should be delayed until y_0's completion (as

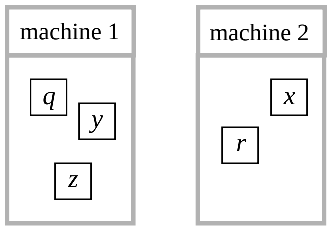


Figure 1.6: Illustration of assignment of methods to machines as defined in Table 1.1.

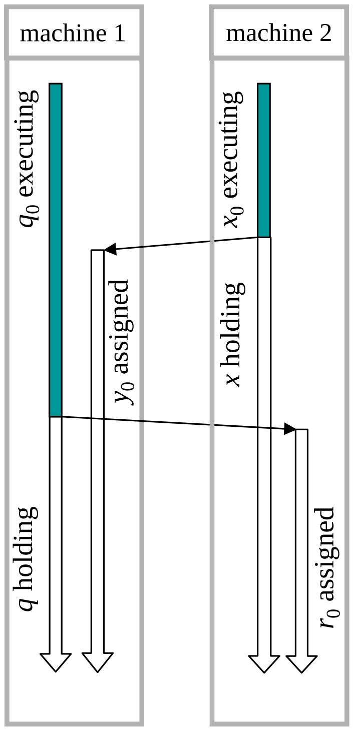


Figure 1.7: Occurrence of deadlock due to the concurrent execution of q_0 and x_0.

in Fig. 1.8) illustrates the fundamental concept of deadlock avoidance. Dijkstra's Banker's Algorithm is the classic approach to deadlock avoidance, and is the foundation of the deadlock avoidance approach developed in this thesis.

In this thesis, a distributed SOA is modeled in which clients submit jobs that have execution deadlines. The execution deadlines define the time by which all service requests with an associated job should be completed. In this thesis, a job is modeled as a workflow graph (WFG), which is a structure that includes high-level control nodes that define execution precedence constraints among its underlying

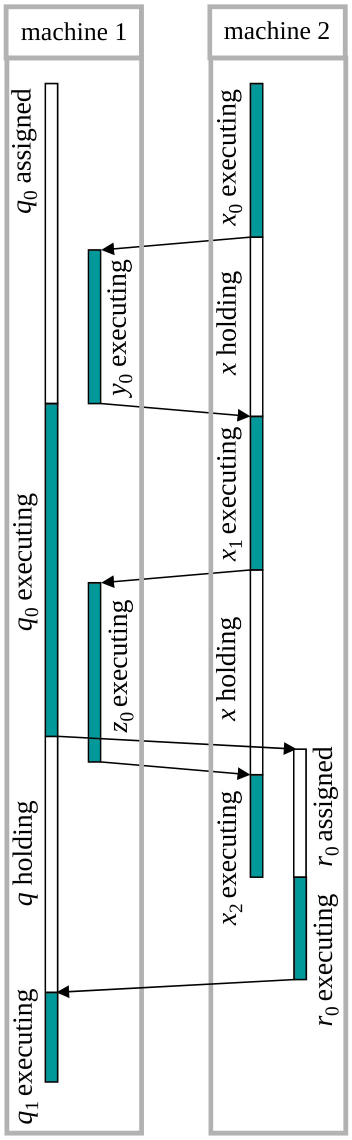


Figure 1.8: Deadlock is avoided by staggering the execution of q_0 and x_0.

service requests. WFGs can vary greatly in size and structure. For example, the simplest WFG is comprised of a single service request. At the other extreme, a large WFG may contain thousands of service requests. In addition, a WFG may be highly parallelizable or it may represent a single chain of service requests in which no two requests may be executed in parallel.

Three types of WFGs are considered in this thesis: Batch, Webservice, and Interactive. Batch WFGs have daily periodicity, which distinguishes them from Interactive and Webservice WFGs. Batch WFGs generally have a more complex structure, consisting of more complex call trees, compared to the other WFG types. In addition, the different WFG types have different arrival processes and deadline characteristics. WFGs are discussed in more detail in Chapter 4.

In the framework considered here, a simulation environment is employed in which WFGs are submitted to a component of the system referred to as `Advancer`. The responsibility of `Advancer` is to assign requests of submitted WFGs to machines of the cluster with a primary objective of assigning requests in such a way to reduce missed deadlines of all WFGs. The simulation environment models each machine in the cluster of memory-managed multicore machines. CPU and memory resources are the two primary factors used to characterize machines. `Advancer` is assumed to have estimates for the CPU and memory requirements of each request in a WFG.

As `Advancer` assigns requests to machines, the simulation environment tracks aggregate CPU and memory loading of each machine based on the CPU and memory requirements of all requests currently executing on the machine. When a request is assigned to a machine, the aggregate CPU and/or memory loading increases, which generally decreases the overall performance of the machine. Likewise, after a request completes execution, the performance of the machine on which the request was executing generally increases as the aggregate loading on the machine decreases. A request may finish a portion of the work to be done but still consume memory resources if it is waiting on child requests to complete. Such a request is said to be in the "holding" state. Requests of this nature originate from an intermediate node in a call tree and can lead to deadlock if appropriate action is not taken. In the framework developed, `Advancer` employs `Banker` to avoid deadlock. `Banker` is a deadlock avoidance technique that is developed in this thesis. `Banker` is an adaptation of Dijkstra's original Bankers Algorithm [11] for a distributed SOA modeled here. In addition to preventing deadlock, it is shown that `Banker` can also improve the performance of sub-optimal policies in terms of workflow tardiness.

The remainder of the thesis is organized in the following manner. Chapter 2 examines the phenomenon of deadlock and describes the motivation for this thesis. Chapter 3 and Chapter 4 outline a Call Tree Execution Model and Workflow Graph model respectively. The `Banker's` algorithm for deadlock avoidance in a distributed SOA is developed in Chapter 5. An overview of the simulation environment is given in Chapter 6, followed by simulation results in Chapter 7, and concluding remarks in Chapter 8.

Chapter 2

Overview of Previous Work on the Deadlock Problem

Deadlock describes an undesirable phenomenon in which two or more processes wait indefinitely. Processes can enter the deadlock state—also known as the deadly embrace [11]—when each process is stalled because it is waiting for resources held by another process to be freed. For the case of two processes, deadlock occurs when the first process is waiting on the second process to free resources that the first process needs in order to proceed; however, the second process is also stalled because it is waiting for resources to be freed by the first. In general, deadlock can involve more than two processes; occurring when the resource requirements of the involved processes form a circular chain of dependencies with one another.

Deadlock can be expressed more precisely with a state-graph [12]. Assume a system contains a set of executing tasks $\{T_1, T_2, \ldots, T_n\}$ each utilizing a distinct resource of the available resources. The system contains one resource instance for each resource type in the set of available resources $\{R_1, R_2, \ldots, R_m\}$. A state graph is constructed with the nodes representing the resources. There is an arc in the state graph from R_j to R_k if at the time instant for which the graph represents, some task T_i occupies resource R_j and requests R_k. It has been shown [12, 13, 11] that a deadlock exists if the state graph contains a cycle. For example, consider Fig. 2.1

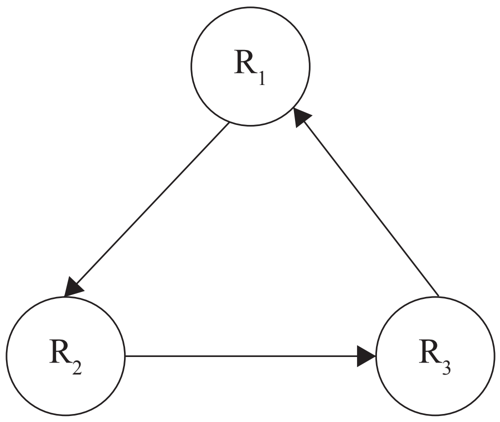


Figure 2.1: Example of a state graph with a cycle indicating deadlock.

which illustrates a system in a deadlocked state.

2.1 Necessary Conditions for Deadlock

The seminal paper [12] defines four necessary conditions for the occurrence of deadlock. The necessary conditions are summarized below.

- Mutual-Exclusion Condition: Tasks claim exclusive use of resources allocated to them.
- Wait-For Condition: Tasks continue to hold resources already allocated to them while waiting for additional resources.
- No-Preemption Condition: Resources cannot be revoked from a task holding them until the task completes and releases them.
- Circular-Wait Condition: A cycle of tasks exists in which each task holds one or more resources that are needed by the next task in the chain.

Even when these conditions are possible, it is generally not straightforward to determine if, or when, deadlock will occur.

2.2 Combating Deadlock

Generally speaking, deadlock can be dealt with in one of three ways: preventative measures can be taken to prevent deadlock from ever occurring, deadlock may be allowed to happen and measures can be taken to recover, or a protocol can be employed to avoid deadlock. The following three sub-sections describe the techniques of deadlock prevention, deadlock detection and recovery, and deadlock avoidance. The approach developed in this thesis is categorized as a deadlock avoidance technique.

2.2.1 Deadlock Prevention

As stated in Section 2.1, for deadlock to occur the four necessary conditions must be satisfied. Fundamental to deadlock prevention is the concept that the system be designed in such a way such that at every point in time at least one of the necessary conditions cannot possibly be satisfied.

The mutual-exclusion condition is one that cannot be denied for all resources as some resources are inherently unsharable. For example, consider the scenario in which multiple process all want to write to the same file. Only one of those processes can be granted permission to write to the file. In this way, some resources simply cannot be shared so the mutual-exclusion condition cannot be removed in all cases [12].

The second of the necessary conditions, the wait-for condition, can be removed if no task is allowed to request resources while it occupies any other resources. One way to achieve this is to force all tasks to request all necessary resources needed for completion before execution begins. An alternative to this approach is to force a task release all occupied resources before requesting additional resources. Both of these protocols have two major disadvantages. First, resource utilization is likely to be low because resources may be allocated to a task that is not using them at the present time. Second, starvation is possible. If a task requires multiple resources, there is no guarantee that all of the resources needed for that task will be free at any given moment in time. If this occurs, the task will be forced to wait for an arbitrarily long period of time [13]. This is known as starvation.

The third condition requires that there be no preemption of resources that have already been allocated to a task. To ensure that this condition is not satisfied, the following protocol can be used. When a task T requests resources that cannot be immediately allocated to it (i.e. the task must wait), then all resources allocated to T are immediately preempted. Then, at some point in the future when all resources previously occupied and newly requested by task T are available, T can resume execution. As an alternative, another protocol may be used. This protocol states that when a task requests resources, a check is done to ensure that adequate resources are available. If adequate resources are available, they are allocated to the requesting task. Otherwise, if another task in the holding state is occupying adequate resources to satisfy the requesting task, the resources are preempted from the holding task. The waiting task will then have to regain the preempted resources before it can continue with its execution. If resources are neither freely available nor held by a waiting task, the requesting task must wait. During the waiting period, the resources held by the requesting task may be preempted if another task requests them. This protocol can only be used with resources whose state can be easily saved and restored. Of course, not all resources meet this criteria. In addition, it may not always be practical to save and restore the state of resources, especially if the system is one in which computational deadlines exist.

The final condition for deadlock, the circular-wait condition, can be removed if a total ordering is imposed on the set of all resource types and each task is required to request resources only in an increasing order of enumeration [13]. For example, consider a system with the following set of resource types $R = \{R_1, R_2, \ldots, R_n\}$. The circular-wait condition is removed if all tasks request resources in only an increasing order. Initially, a task can request any amount of R_i. After that, the task can request resource R_j if and only if the number assigned to R_j in the total ordering is strictly greater than that assigned to R_i. For example, if task T needs both R_1 and R_3 to complete execution, it must request the resources in exactly that order.

2.2.2 Deadlock Detection and Recovery

Sometimes designing a system such that deadlock can never occur is impractical. Another solution to dealing with the deadlock problem is deadlock detection and recovery. For this two pronged approach to be successful, the system must provide an efficient algorithm for detecting when deadlock has occurred and some recovery algorithm that removes the deadlock.

For the case of one resource instance of each resource type, a deadlock detection scheme can be implemented by maintaining and observing a *wait-for* graph [13]. The graph contains an edge from T_i to T_j if T_i is waiting for resources currently occupied by T_j. A deadlock exists in the system if and only if there exists a cycle in the *wait-for* graph. For a deadlock to be detected in the system, the system must track resource allocations in the *wait-for* graph and periodically invoke an algorithm to check for a cycle in the *wait-for* graph.

The *wait-for* graph and cycle detection algorithm works well for detecting deadlock in a system where all resources have only a single instance but is insufficient for systems in which resources may have muliple instances. For such systems, a more complicated approach is needed. Such an approach is discussed in detail in [13] and is summarized here. The algorithm utilizes several data structures that track the amount of available resources, the allocation of resources to each task, and the amount of resources each task is currently requesting. When the algorithm is invoked, it uses these data structures to examine every possible allocation sequence for the tasks that have been started but have yet to complete. If all started tasks can not be completed, deadlock is declared. Of major concern when utilizing a deadlock detection algorithm is the decision of under what conditions to run the detection algorithm. If the nature of the system is such that deadlocks occur frequently, the detection algorithm should be invoked frequently. Another factor that should be considered is the amount of overhead required to run the algorithm. If this overhead is extremely high, it may be infeasible to run the detection algorithm as often as necessary.

Deadlock detection algorithms are alone not enough to solve the deadlock problem. A detection algorithm must be paired with some form of a recovery algorithm whose responsibility is to rid the system of all deadlocks. Recovering from deadlock is a domain specific problem and can be very complex. For example, in the case of a system where tasks must be completed before hard deadlines, the recovery algorithm should find the most efficient way to recover from deadlocks while not causing any more deadlines to be missed than are absolutely necessary. In general, two main approaches are considered to recover from deadlock. The first approach is to abort all tasks involved in the deadlock. Of course, this approach will clear the deadlock but will also most likely terminate tasks that need not be terminated. This is inefficient, especially if long-running tasks are involved because any partial computations will have to be recomputed. Another, seemingly more efficient, approach involves aborting individual processes until the deadlock cycle is removed. The problem with this approach is that a deadlock detection algorithm must be invoked after the termination of each individual task. This too can lead to inefficiencies. Sometimes the costs associated with a detection and recovery scheme are simply too great to be feasible. An example of a real-world system in which this is the case is BlueBox described in Section 1.4. In BlueBox, jobs submitted by clients have specific service level agreements (SLAs) [14] that define agreed upon deadlines for each job. Failing to meet these SLAs results in penalties.

2.2.3 Deadlock Avoidance

An alternative approach to both deadlock prevention and deadlock detection and recovery is deadlock avoidance. Critical to the success of deadlock avoidance is some advance information indicating how tasks will request resources in the future. This information can then be used to determine for each request whether or not serving the request, thereby allocating resources, will put the system in a state of possible deadlock. The amount of information required varies depending on the model used by the specific deadlock avoidance algorithm. The simplest model, and the one employed by the algorithm presented in this thesis, makes use of worst case estimates of resource requirements for each request. Given the information from the model, a deadlock avoidance algorithm can ensure that at every point in time, the circular-wait condition necessary for deadlock is not satisfied by ensuring the system remains in a "safe" state.

A "safe" state is one in which deadlock will not occur. An "unsafe" state does not always lead to deadlock, but indicates that the possibility for deadlock exists. Fig. 2.2 illustrates the relationship between "unsafe" states and deadlock. From the figure, note that the system could operate in the "unsafe" region and never encounter deadlock. However, if the system remains in the "safe" region, deadlock can never occur. Whenever a task makes a request for resources, a deadlock avoidance algorithm will utilize the knowledge it possesses of the current system coupled with knowledge of future estimates of resource requirements to determine whether or not granting the resources to the requesting task will put the system in a "safe" or "unsafe" state. If the resulting state is "safe" the resources can be granted to the requesting task. Otherwise, the requesting task must wait until adequate resources become available that keep the system in a "safe" state.

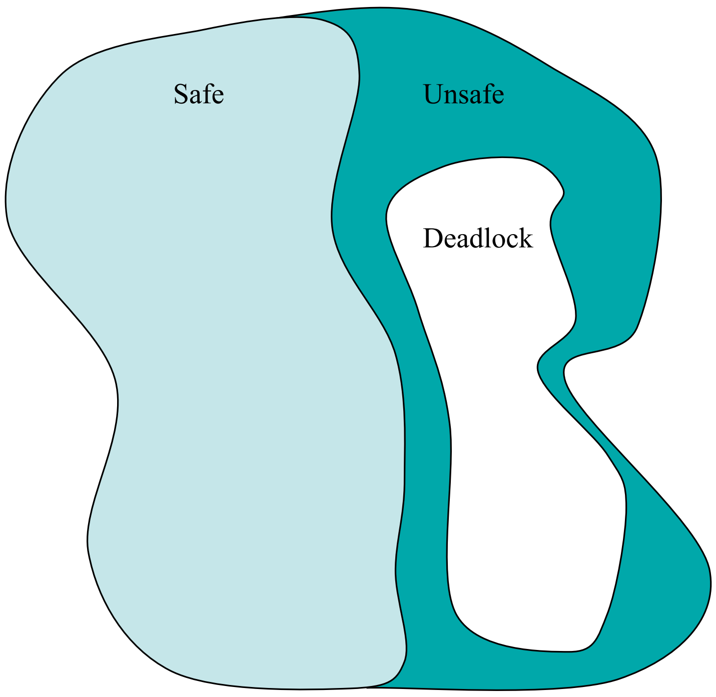


Figure 2.2: Safe, unsafe, and deadlock regions.

A state is said to be “safe” when it is possible, using the currently available resources and those that will be returned by currently executing tasks, to form a “safe sequence” [13] (consisting of all tasks in the system) such that all tasks in the sequence complete. It is important to note that the initial state (when no resources have been allocated) is a “safe” state because a valid sequence can always be formed by executing all tasks in a serial fashion. It follows from this observation that if a “safe sequence” can be determined for currently executing tasks, remaining tasks that have yet to start can always be run to completion [12]. It is the job of the employed deadlock avoidance algorithm to ensure that the system is always in a “safe” state by discovering a “safe sequence” each time a request for resources is made.

An algorithm for deadlock avoidance in the context of operating systems is given in [15]. In addition, an algorithm for deadlock avoidance in distributed systems is given in [16]. While both of these algorithms are based upon extensions of Dijkstra’s

Banker's Algorithm, neither address certain complexities inherent to an SOA. For example neither handle integration into an SOA at the service orchestration layer or facilitate the dynamic assignment of requests by a SelectionPolicy. For these reasons, a novel approach is developed in this thesis.

Chapter 3

Call Tree Execution Model

The execution of call tree T of Fig. 3.1 involves calls to six service methods, labeled a through e. A method is a sequence of one or more method segments; the boundaries between segments are defined where a method calls another method. Also shown in the figure is an expanded view of T, illustrating the sequential ordering in which the methods' segments are interleaved. Thus, during execution of a call tree, at most one of its methods is executing at a time. Methods that do not call other methods are the single-segment leaves of the tree, e.g., b_0, d_0, e_0, and f_0 in the figure. The non-leaf methods (a and c in the figure) each include calls to other methods and thus are comprised of multiple segments.

Once a method's initial segment is assigned to a machine, it is assumed that its subsequent segments are pre-assigned to the same machine, i.e., method migration is not considered. Thus, the segments subsequent to the initial segment are shaded in Fig. 3.1 to indicate that their assignment is inherited from the assignment of the initial segment of the method.

3.1 Method Segment States

Fig. 3.2 illustrates the sequential ordering (vertically) in which the chain of segments of call tree T of Fig. 3.1 are executed. The possible states of the segments are defined

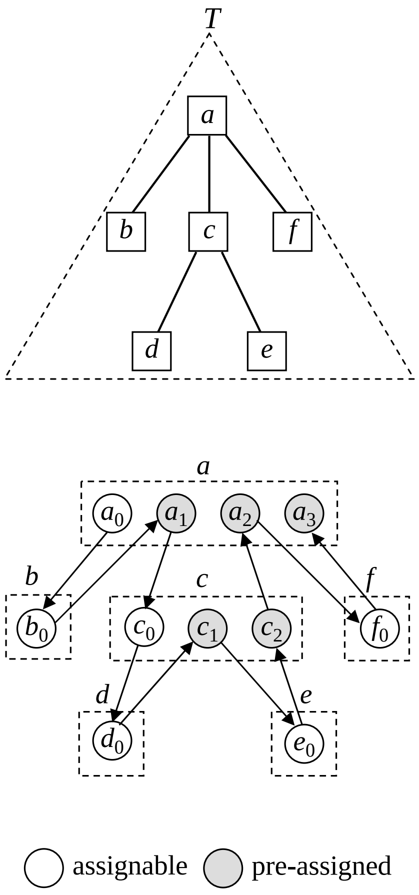


Figure 3.1: Example call tree T and expanded view illustrating sequential ordering of its method segments.

by the labeled columns: blocked (B); ready (R); executing (E); and completed (C). Upon initialization of call tree T, segment a_0 transitions from blocked to ready, and all other segments remain blocked. After zero or more time units in the ready state, a_0 is assigned to a machine (by some independent assignment policy) and begins executing, which is represented by a_0's transition from the ready state to the executing state.

Segment a_0 stays in the executing state until the segment completes execution. The completed state is a terminal state for a segment. The dashed transition arc emanating from the completed state signifies that the completion of a segment triggers the next segment in the chain to transition out of the blocked state. For the

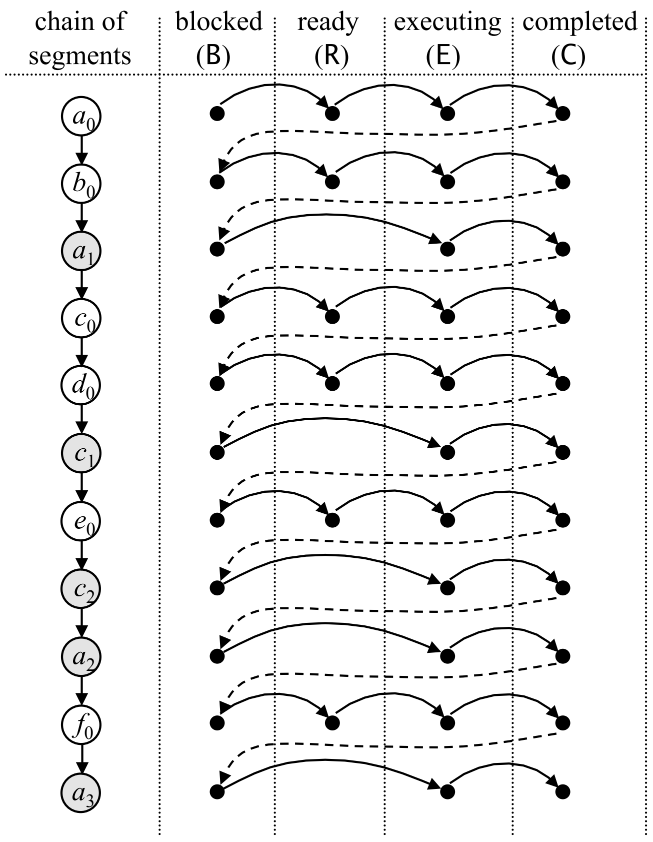


Figure 3.2: Execution ordering of method segments of call tree T from Fig. 3.1 (shown vertically) with state transition diagram for each segment (shown horizontally).

transition from a_0 to b_0, note that b_0 moves from blocked to ready upon a_0's completion. However, upon b_0's completion, the next segment, a_1, transitions directly from blocked to executing, bypassing the ready state. This is because a_1 is pre-assigned according to a_0's assignment. Fig. 3.2 incudes the state transition diagram for the entire execution of T of Fig. 3.1.

In general, a segment g_i is in exactly one defined state at a time. Let $\sigma(g_i)$ denote a mapping from a segment g_i to a representation of its state. Making use of the abbreviations for the states provided in Fig. 3.2, the set of possible states for an *initial* segment g_0 is defined by:

$$\sigma(g_0) \in \{\mathsf{B}, \mathsf{R}, \mathsf{E}, \mathsf{C}\}. \tag{1}$$

Similarly, the set of possible states for a *subsequent* segment g_i, $i > 0$ is defined by:

$$\sigma(g_i) \in \{\mathsf{B}, \mathsf{E}, \mathsf{C}\}. \tag{2}$$

The R (ready) state is not a possible state for a subsequent segment because a subsequent segment inherits its assignment from its associated initial segment; thus, it transitions directly from B to E.

The total number of segments associated with a call tree is an important parameter, and shall be denoted by N_T. For the call tree T of Fig. 3.1, $N_T = 11$. In general, the value of N_T is the sum of the number of vertices and edges in the call tree graph, which for T of Fig. 3.1 is $N_T = 6 + 5$.

The two time instances when each of the N_T segments of a call tree begin executing, and complete executing, are important markers in defining life cycle phases of a call tree. Define t_{E_i} as the i^{th} *begin execution time marker* for a call tree, which is the time instant when the i^{th} segment of a call tree begins executing. Similarly, define t_{C_i} as the i^{th} *completion time marker* for a call tree, which is the time instant when the i^{th} segment of a call tree completes executing.

To illustrate examples of execution and completion markers, consider call tree T of Fig. 3.1, with its corresponding chain of $N_T = 11$ segments shown in Fig. 3.2. The instant that the fourth segment c_0 begins executing, i.e., the instant that $\sigma(c_0)$ transitions from R to E, defines t_{E_4}. Similarly, t_{C_4} is defined as the instant that c_0 completes executing, i.e., the instant that $\sigma(c_0)$ transitions from E to C.

3.2 Method States

In general, a method g can be in one of five possible states. Four of the states for a method g are named the same as the four possible states of a segment, i.e., $\{\mathsf{B}, \mathsf{R}, \mathsf{E}, \mathsf{C}\}$. The definitions for these states for method g follow logically from the states of g's underlying segments.

$$\sigma(g) = \mathsf{B} \Leftrightarrow \sigma(g_0) = \mathsf{B} \tag{3}$$

$$\sigma(g) = \mathsf{R} \Leftrightarrow \sigma(g_0) = \mathsf{R} \tag{4}$$

$$\sigma(g) = \mathsf{E} \Leftrightarrow \exists i \text{ s.t. } \sigma(g_i) = \mathsf{E} \tag{5}$$

$$\sigma(g) = \mathsf{C} \Leftrightarrow \sigma(g_i) = \mathsf{C}, \forall i \tag{6}$$

In addition to the four states defined above, a method has another possible state called holding (H), defined as follows:

$$\sigma(g) = \mathsf{H} \Leftrightarrow \begin{cases} \sigma(g_0) = \mathsf{C} & \& \\ \exists i \text{ s.t. } \sigma(g_i) \neq \mathsf{C} & \& \\ \sigma(g_i) \neq \mathsf{E}, \forall i > 0 & \end{cases} \tag{7}$$

As an example of method a in Fig. 3.1 in the holding state, i.e., $\sigma(a) = \mathsf{H}$, consider the following states of a's segments: $\sigma(a_0) = \mathsf{C}$, $\sigma(a_1) = \mathsf{C}$, $\sigma(a_2) = \mathsf{B}$, and $\sigma(a_3) = \mathsf{B}$. This represents a state in which a is partially completed, and none of a's segments are executing. From the structure of the call tree, it is apparent that this state for a implies that $\sigma(b_0) = \mathsf{C}$, and that only c, d, or e could possibly be executing.

Based on the definition of the holding state provided by Eq. 7, a method that is a leaf of a call tree can never be in the holding state. To show this, recall that a leaf method has only one segment, e.g., g_0. Thus, it is not possible for such a method to be in the holding state because it is not possible to satisfy the two conditions $\sigma(g_0) = \mathsf{C}$ & $\sigma(g_0) \neq \mathsf{C}$, refer to Eq. 7.

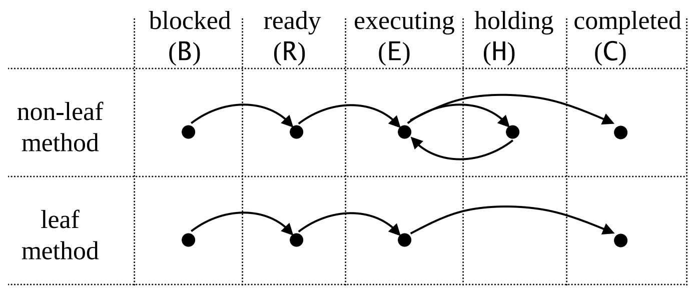


Figure 3.3: State transition diagrams for non-leaf and leaf methods.

Fig. 3.3 illustrates the state transition diagrams for both non-leaf and leaf methods. For non-leaf methods, notice the presence of the cycle involving states E and H. The state transition for a leaf method contains no cycle; its transitions are the same as the initial segment transitions illustrated previously in Fig. 3.2.

3.3 Life Cycle Phases of a Call Tree

Recall from Subsection 3.1 that a call tree T has N_T begin execution time markers and N_T completion time markers, which define time instances when each of the N_T segments of a call tree begin and complete execution, respectively. These markers are denoted by t_{E_i} and t_{C_i}, and are used here in partitioning the time line of a call tree's life cycle into $N_T + 2$ phases, numbered $0, 1, \ldots, N_T + 1$. The i^{th} *phase* of a call tree's life cycle is denoted by ϕ_i and defined by

$$\phi_i = \begin{cases} [0, t_{\mathsf{E}_1}), & i = 0 \\ [t_{\mathsf{E}_i}, t_{\mathsf{E}_{i+1}}), & i \in \{1, 2, \ldots, N_T - 1\} \\ [t_{\mathsf{E}_i}, t_{\mathsf{C}_i}), & i = N_T \\ [t_{\mathsf{C}_i}, \infty), & i = N_T + 1 \end{cases} \tag{8}$$

Phases ϕ_i, $i \in \{1, 2, \ldots, N_T - 1\}$, can be further partitioned into two sub-phases called the i^{th} *primary sub-phase*, denoted ϕ_i^1, and the i^{th} *secondary sub-phase*, denoted ϕ_i^2:

$$\phi_i^1 = [t_{\mathsf{E}_i}, t_{\mathsf{C}_i}), i \in \{1, 2, \ldots, N_T - 1\} \tag{9}$$

$$\phi_i^2 = \left[t_{\mathsf{C}_i}, t_{\mathsf{E}_{i+1}}\right), i \in \{1, 2, \ldots, N_T - 1\} \tag{10}$$

From Eqs. 8 through 10, it follows that

$$\phi_i = \phi_i^1 \cup \phi_i^2, i \in \{1, 2, \ldots, N_T - 1\} \tag{11}$$

and

$$\phi_i^1 \cap \phi_i^2 = \emptyset, i \in \{1, 2, \ldots, N_T - 1\} \tag{12}$$

The primary sub-phase ϕ_i^1 represents the time interval when segment i is executing. The secondary sub-phase ϕ_i^2 represents the time interval after segment i has completed execution, but before segment $i+1$ begins executing. Thus, ϕ_i^2 represents a "delay" time in which neither segment i nor segment $i+1$ is executing.

In general, the length of ϕ_i^2 is impacted by the characteristics of the underlying scheduling policy. To illustrate, consider the chain of segments in Fig. 3.2. The completion of the 4^{th} segment (c_0) marks the value of t_{C_4} and triggers the 5th segment (d_0) to transition from state B to R. Once d_0 is assigned, d_0 transitions to state E. The underlying scheduling policy dictates when d_0 should begin executing, i.e., transition from state R to state E, which defines the value t_{E_5} as well as the length of $\phi_4^2 = [t_{\mathsf{C}_4}, t_{\mathsf{E}_5})$.

3.4 Resource Requirements

A method g that is in the executing state (i.e, $\sigma(g) = \mathsf{E}$) is assumed to require both memory and CPU resources. A method g in the holding state (i.e., $\sigma(g) = \mathsf{H}$) is assumed to require only memory resources. A method in a state other than E or H

is assumed to have no resource requirement.

The notation $T^{(i)}$ is used to indicate that call tree T is in life cycle phase ϕ_i. Furthermore, $T_{\mathsf{E}}^{(i)}$ denotes the set of executing methods associated with $T^{(i)}$:

$$T_{\mathsf{E}}^{(i)} = \{g \in T^{(i)} : \sigma(g) = \mathsf{E}\}. \tag{13}$$

Because at most one method can execute at a time, $T_{\mathsf{E}}^{(i)}$ either contains zero or one elements.

Similarly, $T_{\mathsf{EH}}^{(i)}$ denotes the set of methods of $T^{(i)}$ that are either executing or holding:

$$T_{\mathsf{EH}}^{(i)} = \{g \in T^{(i)} : (\sigma(g) = \mathsf{E}) \vee (\sigma(g) = \mathsf{H})\}. \tag{14}$$

The resource requirements of a call tree change when it transitions from one phase of its life cycle to the next. To illustrate, Table 3.1 defines the sets $T_{\mathsf{E}}^{(i)}$ and $T_{\mathsf{EH}}^{(i)}$ for each of the twelve phases of call tree T of Fig. 3.1. Technically, the sets provided in the table are associated with T's primary sub-phases, ϕ_i^1 for $i \in \{1, 2, \ldots, 11\}$. During T's i^{th} secondary sub-phase (ϕ_i^2) $T_{\mathsf{E}}^{(i)} = \emptyset$ because ϕ_i^2 defines the period of time after completion of segment i but before beginning the execution of segment $i + 1$.

Table 3.1: Life cycle phases, ϕ_i, for call tree T of Fig. 3.1. Also tabulated are sets of methods requiring CPU resources, $T_{\mathsf{E}}^{(i)}$, and sets of methods requiring memory resources, $T_{\mathsf{EH}}^{(i)}$, during each phase.

i	ϕ_i	$T_{\mathsf{E}}^{(i)}$	$T_{\mathsf{EH}}^{(i)}$
0	$[0, t_{\mathsf{E}_1})$	$\emptyset$	$\emptyset$
1	$[t_{\mathsf{E}_1}, t_{\mathsf{E}_2})$	$\{a\}$	$\{a\}$
2	$[t_{\mathsf{E}_2}, t_{\mathsf{E}_3})$	$\{b\}$	$\{a, b\}$
3	$[t_{\mathsf{E}_3}, t_{\mathsf{E}_4})$	$\{a\}$	$\{a\}$
4	$[t_{\mathsf{E}_4}, t_{\mathsf{E}_5})$	$\{c\}$	$\{a, c\}$
5	$[t_{\mathsf{E}_5}, t_{\mathsf{E}_6})$	$\{d\}$	$\{a, c, d\}$
6	$[t_{\mathsf{E}_6}, t_{\mathsf{E}_7})$	$\{c\}$	$\{a, c\}$
7	$[t_{\mathsf{E}_7}, t_{\mathsf{E}_8})$	$\{e\}$	$\{a, c, e\}$
8	$[t_{\mathsf{E}_8}, t_{\mathsf{E}_9})$	$\{c\}$	$\{a, c\}$
9	$[t_{\mathsf{E}_9}, t_{\mathsf{E}_{10}})$	$\{a\}$	$\{a\}$
10	$[t_{\mathsf{E}_{10}}, t_{\mathsf{E}_{11}})$	$\{f\}$	$\{a, f\}$
11	$[t_{\mathsf{E}_{11}}, t_{\mathsf{C}_{11}})$	$\{a\}$	$\{a\}$
12	$[t_{\mathsf{C}_{11}}, \infty)$	$\emptyset$	$\emptyset$

Chapter 4

Workflow Graph (WFG)

Workflow graphs (WFGs) are used to model computational jobs submitted to an SOA by users. A WFG is a structure that is a generalization of the call tree structure defined in Chapter 3. Although a single call tree is itself a special case of a WFG, a WFG generally includes multiple call trees.

A WFG includes high-level control nodes that define execution precedence constraints among its member call trees. Two types of control nodes are considered here: *parallel* and *sequential*. A parallel control construct indicates that its child call trees may be executed concurrently. A sequential control construct indicates that its child call trees must be executed in sequence (left to right order).

Workflows of the BlueBox system are written in the Gozer [17] workflow language. In practice, these workflows contain client-specific customizations but generally consist of a structure close to the structure of the sample workflow shown in Fig. 4.2. Fig. 4.1 outlines pseudocode associated with the structure of the sample workflow.

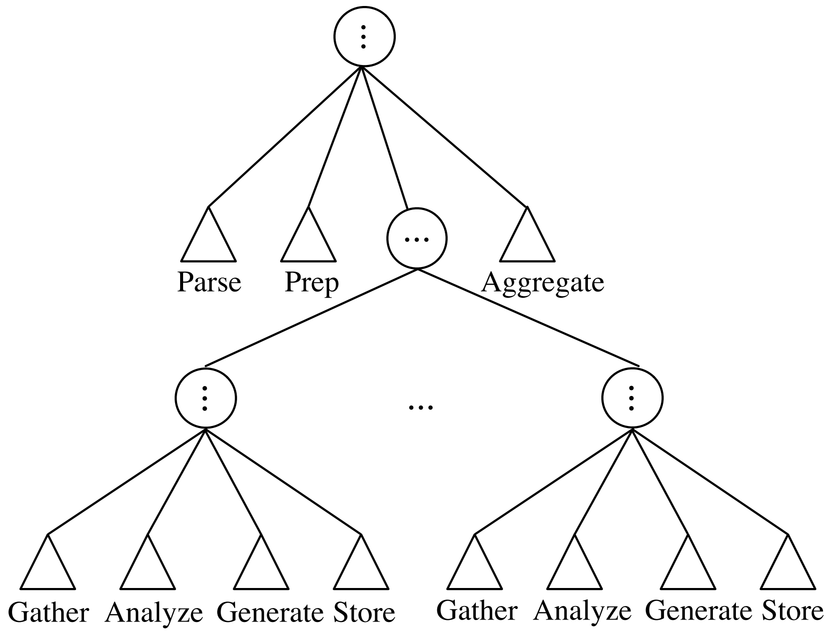
Fig. 4.2(a) shows an example WFG containing eleven call trees, that are represented with triangular shapes. The circular nodes in Fig. 4.2(a) are control nodes that indicate that their direct children can be executed in parallel, represented by nodes labeled with horizontal dots ($\cdots$), or must be executed sequentially, represented by nodes labeled with vertical dots ($\vdots$). Fig. 4.2(b) is a representation of

```
begin workflow
  parse configuration
  prepare report templates
  for each report to run
    gather portfolios
    run analysis
    generate report
    store report
  aggregate report outputs
end workflow
```

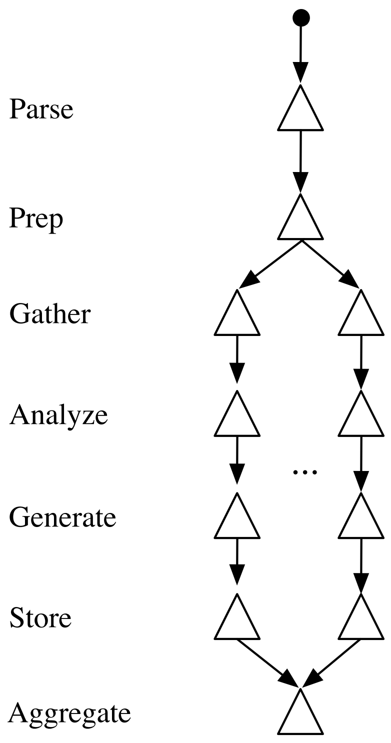
Figure 4.1: Pseudocode for a basic Gozer workflow.

the WFG of Fig. 4.2(a) as a directed acyclic graph in which the directed arcs denote precedence constraints among the call trees.

As described in Chapter 3, call trees are composed of service methods. When a call tree of Fig. 4.2 is executed, a series of service methods is invoked in order to complete the execution of the call tree. To illustrate, Fig. 4.3 defines the service methods invoked when the call tree labled "Analyze" of Fig. 4.2 executes. The root of the "Analyze" call tree is denoted by method AnalyzeMain(AM). The first step in the execution of the "Analyze" call tree is a call to a service known as SecurityManger (SM). This service is responsible for authentication, authorization, and access control. After a successful authentication, the report query will be sent to an external risk engine called RiskServer. Communication to RiskServer occurs through the RSQueryService (RS). The final step of the "Analyze" call tree is a call to the AccontingService (AS), which is repsonsible for various bookkeeping and statistic collection tasks. The statistics collecting step again requires authentication so another call to SM is required. When the call to AS returns to AM, the execution for the "Analyze" tree is complete and the call tree will transition to the completed state.



(a)



(b)

Figure 4.2: (a) Example WFG derived from the pseudocode of Fig. 4.1 with parallel $(\cdots)$ and sequential $(\,\vdots\,)$ control nodes. (b) Representation of the WFG of (a) as a directed acyclic graph, illustrating the precedence constraints among the call trees.

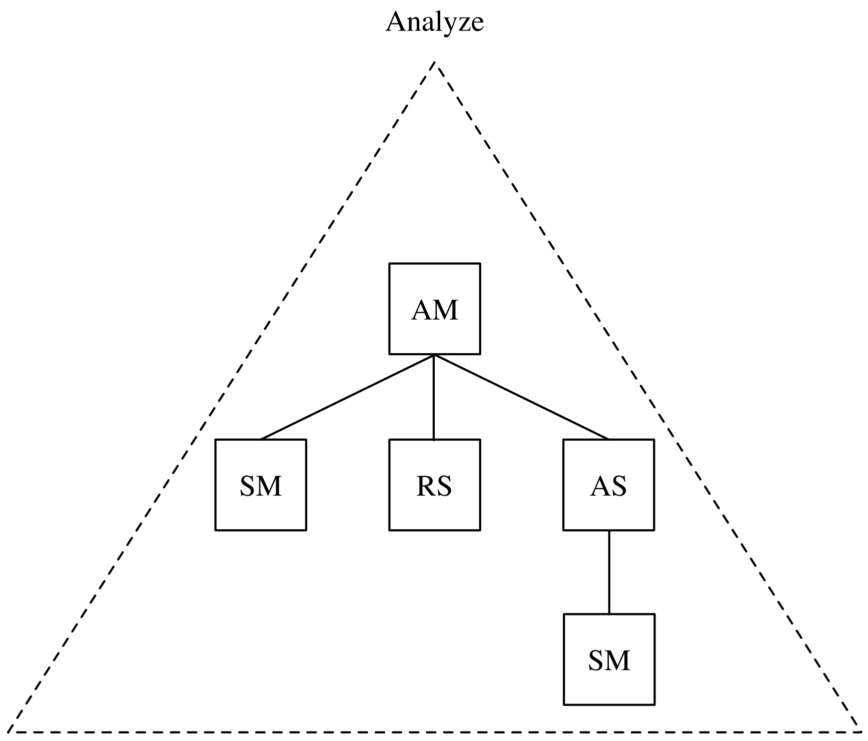


Figure 4.3: An expanded view of the “Analyze” call tree in Fig. 4.2.

Chapter 5

Deadlock Avoidance Algorithm

5.1 Dijkstra's Banker's Algorithm

The classic approach to deadlock avoidance in a resource-constrained environment is Dijkstra's Banker's algorithm [11]. In the original formulation, Banker is assumed to have control of a finite amount of capital (resource), which may be loaned to borrowers. Banker is responsible for accepting or denying (postponing) loan requests made by borrowers. Each borrower has a maximum loan limit, which bounds the total amount of capital each borrower may owe Banker at any instant. In addition to receiving loans, borrowers may pay back all or part of their outstanding loan balance to Banker. A key assumption is this: once a borrower's loan amount reaches its maximum, the borrower will repay this amount back to Banker in finite time.

Banker is tasked with not accepting loan request combinations that could lead to an "unsafe" state (see Subsection 2.2.3). To illustrate, consider the following scenario: Banker has \$9 of capital to loan; Borrower 1 has a credit limit of \$7; and Borrower 2 has a credit limit of \$5. Suppose that Borrower 1 requests a loan for \$5 and Borrower 2 requests a loan for \$3. Although Banker has sufficient capital (\$9) to accept both loan requests (\$8 = \$5 + \$3), doing so would be unsafe because the \$1 remaining with Banker would not be sufficient to cover either borrowers' maximum credit limit. Thus, if either borrower was to request an additional loan of more than \$1 (before either borrower was to pay back any of their current loan) then the result is deadlock.

To illustrate a request scenario involving Borrower 1 and Borrower 2 that would be safe for Banker to grant, suppose Borrower 1 requests a loan for \$5 and Borrower 2 requests a loan for \$2. Granting both of these requests is safe because Banker is left with \$2, which is sufficient to cover Borrower 1's maximum limit (\$7 = \$5 + \$2). Thus, when Borrower 1 (eventually) reaches this max limit, all \$7 will then (in finite time) be paid back to Banker, which provides sufficient capital for Banker to cover Borrower 2's maximum limit. Thus, in this case Banker can guarantee the existence of acceptable (safe) future loan/repayment events to avoid deadlock.

5.2 Banker's Algorithm for a Distributed SOA

The Banker's Algorithm is used here as a basis for avoiding deadlock when scheduling the execution of call trees in a distributed SOA. Assumed to be known to Banker is a phase number for each call tree and the requirements (e.g., memory and CPU) of the methods associated with the call trees. Having access to such information is realistic in the assumed environment in which off-line profiling and/or historical logging can be performed to collect/estimate these requirements. In this context, each call tree is a borrower of resources from the machines of the distributed SOA. As illustrated through the example presented in Section 1.5, deadlock can occur if too many methods from distinct call trees are allowed to advance their phase concurrently (refer to Figs. 1.5 – 1.7); but deadlock can be avoided if proper phasing (i.e., timing for the execution of the underlying methods) is employed, refer to Fig. 1.8.

Unlike the implicit assumption of a centralized pool of resources — used to describe the original Banker's Algorithm — the resources in an SOA are distributed across multiple machines. Thus, when applying Banker's Algorithm in this context, an accounting must be made for the machine location(s) associated with available resources.

Let `Banker` denote the implementation of the Banker's Algorithm developed here for avoiding deadlock in a distributed SOA. In this context, `Banker` serves as a consultant to `Advancer`, which is the orchestration component of the system responsible for incrementing the phases of "advance-able" call trees in order to meet desired system objectives. Examples of such objectives might include maximizing throughput, minimizing latency, and/or minimizing WFG tardiness. The role of `Banker` is to declare whether a proposed phase advancement is safe. In order to meet desired system objective(s), `Advancer` relies on an underlying `SelectionPolicy` module. Thus, `Advancer` first consults with `SelectionPolicy` to obtain a proposed phase advancement, and then consults with `Banker` to determine if it is safe to implement the (proposed) phase advancement. In its attempt to meet desired objectives, `SelectionPolicy` may propose the execution of many call trees and/or aggressively advance the phasing of many call trees. Before committing to the proposed advancement, `Advancer` consults with `Banker` to determine safety. If `Banker` declares the proposed phasing to be `UNSAFE`, then `Advancer` again calls on upon `SelectionPolicy` to produce a modified phasing until one is determined that `Banker` declares to be `SAFE`. The interaction between `Advancer`, `SelectionPolicy`, and `Banker` is provided in Fig. 5.2. The notation and definitions used in the pseudocode of Fig. 5.2 are provided in Fig. 5.1.

From Fig. 5.2 `Advancer` takes as input the set of call trees under consideration, along with their current phasing, and the set of machines. At line 1, a temporary set $\widetilde{\mathbf{T}}$ is constructed containing all trees that do not have a method executing; these represent trees that are eligible for phase advancement, i.e., they are advance-able. `SelectionPolicy` selects a tree, T^*, and a machine assignment, M^*, on which T^*'s

$\mathbf{T}$:	set of all call trees
$\mathbf{I}$:	current phasing; maps trees to phase numbers $\mathbf{I} : \mathbf{T} \rightarrow \{0, 1, \ldots, N_{\max}\}$ where $N_{\max} = \max\{N_T : T \in \mathbf{T}\}$
$\mathbf{I}^*$:	proposed phasing; maps trees to phase numbers
$\mathbf{M}$:	set of machines where methods assigned to each $M \in \mathbf{M}$ are associated with current phasing $\mathbf{I}$
$\mathbf{M}^*$:	set of machines where methods assigned to each $M \in \mathbf{M}$ are associated with proposed phasing $\mathbf{I}^*$

Figure 5.1: Notation and definitions used by pseudocode of Fig. 5.2

```
Advancer($\mathbf{T}, \mathbf{I}, \mathbf{M}$)
1  $\widetilde{\mathbf{T}} \leftarrow \{T \in \mathbf{T} : T_E \neq \emptyset\}$
2  while($\widetilde{\mathbf{T}} \neq \emptyset$)
3    ($T^*, g^*, M^*$)$\leftarrow$ SelectionPolicy($\widetilde{\mathbf{T}}, \mathbf{I}, \mathbf{M}$)
4    if(!$M^*$)
5      break
6    $\mathbf{I}^* \leftarrow \mathbf{I} - \{(T^*, \mathbf{I}(T^*))\} + \{(T^*, \mathbf{I}(T^*) + 1)\}$
7    $\mathbf{M}^* \leftarrow$ update($g^*, M^*, \mathbf{M}$)
8    if(Banker($\mathbf{T}, \mathbf{I}^*, \mathbf{M}^*$)=SAFE)
9      $\mathbf{I} \leftarrow \mathbf{I}^*$
10     $\mathbf{M} \leftarrow \mathbf{M}^*$
11   $\widetilde{\mathbf{T}} \leftarrow \widetilde{\mathbf{T}} - \{T^*\}$
```

Figure 5.2: `Banker` used as a consultant by `Advancer`.

ready method g^* should execute (line 3). Assuming an assignment is made, a corresponding proposed phasing is constructed by incrementing the phase number of the selected tree (line 6). Likewise, the proposed states of the machines are updated by assigning ready method g^* to machine M^* (line 7). `Banker` is then consulted to determine safety of the proposed phasing and associated machine states. If safety is declared by `Banker`, then the proposed phasing and machine states are committed (lines 8 through 10). The selected tree T^* is removed from the temporary set $\widetilde{\mathbf{T}}$ (line 11) and execution continues until $\widetilde{\mathbf{T}} = \emptyset$ (line 2) or `SelectionPolicy` does not return a machine assignment (lines 4 and 5).

The pseudocode for `Banker` is provided in Fig. 5.4. `Banker` returns a value of either `SAFE` or `UNSAFE`. By returning `SAFE`, `Banker` asserts that given the call trees' proposed phase numbers, there exists future scheduling phases to complete execution (without deadlock) of all call trees in $\mathbf{T}$. By returning `UNSAFE`, `Banker` declares that continuing execution of call trees from their current state could lead to deadlock.

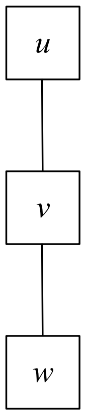


Figure 5.3: Three-level aggregate tree structure associated with tree T of Fig. 3.1.

Assumed to be known for each tree phase is a worst-case estimate of future resource requirements for executing the tree to completion. This estimate is modeled by an *aggregate tree* and denoted by $\mathbf{A}(\mathbf{T}^{(\mathbf{I}^*(\mathbf{T}))})$. The depth of an aggregate tree equals the depth of the original tree and the number of methods comprising an aggregate tree equals its depth (i.e., it is a linear tree). For example, an aggregate tree associated with tree T of Fig. 3.1 has three methods and a depth of three. The resource requirements of each method of an aggregate tree equals the maximum resource requirements of methods yet to be completed at that level. Shown in Fig. 5.3 is the structure of the aggregate tree associated with tree T of Fig. 3.1. For phase 0 (before beginning execution of segment a_0) the resource requirements of the associated aggregate tree are as follows: $R(u) = R(a)$; $R(v) = \max\{R(b), R(c), R(f)\}$; $R(w) = \max\{R(d), R(e)\}$. For phase 3 (after completion of segment a_1 and before beginning execution of c_0) the resource requirements of the associated aggregate tree are as follows: $R(u) = 0$; $R(v) = \max\{R(c), R(f)\}$; $R(w) = \max\{R(d), R(e)\}$.

From Fig. 5.4, `Banker` takes as input the set of call trees under consideration, along with their proposed phasing, and the set of machines with proposed method assignments. At line 1, a temporary copy of $\mathbf{T}$ is constructed, denoted by $\widehat{\mathbf{T}}$. For each tree, Banker determines if sufficient resources are available to satisfy future resource requirements for that tree. This is accomplished by employing `SelectionPolicy` to assign a machine to each method of the associated aggregate tree (lines 7 and 8). If all methods of the aggregate tree are assigned to a machine, then resources consumed by the tree are released (line 14). Also, the tree is removed from $\widehat{\mathbf{T}}$. Banker iterates over $\widehat{\mathbf{T}}$ until either: $\widehat{\mathbf{T}}$ is empty, in which case Banker returns `SAFE` (line 19) or all remaining trees' future resource requirements cannot be satisfied (lines 17 and 18).

```
Banker($\mathbf{T}, \mathbf{I}^*, \mathbf{M}^*$)
1  $\widehat{\mathbf{T}} \leftarrow \mathbf{T}$
2  while($\widehat{\mathbf{T}} \neq \emptyset$)
3    aTreeFits $\leftarrow$ FALSE
4    for($T \in \widehat{\mathbf{T}}$)
5      futurePhasesFit $\leftarrow$ TRUE
6      $\widehat{\mathbf{M}} \leftarrow \mathbf{M}^*$
7      for($g \in A(T^{(\mathbf{I}^*(T))})$)
8        $M^* \leftarrow$ SelectionPolicy($\{g\}, 0, \widehat{\mathbf{M}}$)
9        if(!$M^*$)
10         futurePhasesFit $\leftarrow$ FALSE
11         break
12       $\widehat{\mathbf{M}} \leftarrow$ update($g, M^*, \widehat{\mathbf{M}}$)
13     if(futurePhasesFit)
14       $\mathbf{M}^* \leftarrow$ release($T^{(\mathbf{I}^*(T))}, \mathbf{M}^*$)
15       aTreeFits $\leftarrow$ TRUE
16       $\widehat{\mathbf{T}} \leftarrow \widehat{\mathbf{T}} - \{T\}$
17   if(!aTreeFits)
18     return UNSAFE
19 return SAFE
```

Figure 5.4: `Banker`: Dijkstra's Banker's Algorithm for a distributed SOA.

Chapter 6

Simulation Environment

6.1 Overview

Fig. 6.1 illustrates the major components of the simulation environment considered in this thesis. The framework considered here models a real-world service oriented architecture (SOA) known as BlueBox currently in use at RiskMetrics Group. Clients of RiskMetrics Group submit workflows to BlueBox in a number of different ways. The workflows are then executed and the results are sent back to the clients. In the simulation environment, clients' workflows are modeled by WFG Generator, shown in the figure. The WFG Generator generates WFGs that are one of three types: Interactive, Batch, and Webservice. After the WFGs are generated, they are sent to Advancer to be executed.

Advancer is responsible for the execution of the generated WFGs on the cluster of machines. The generated WFGs are composed of call trees. As described in Chapter 3, call trees are composed of service methods. An invoked service method is synonymous with a service request in an SOA and is referred to simply as a request for the remainder of this chapter.

Advancer relies on Selection Policy to determine exactly what and where requests

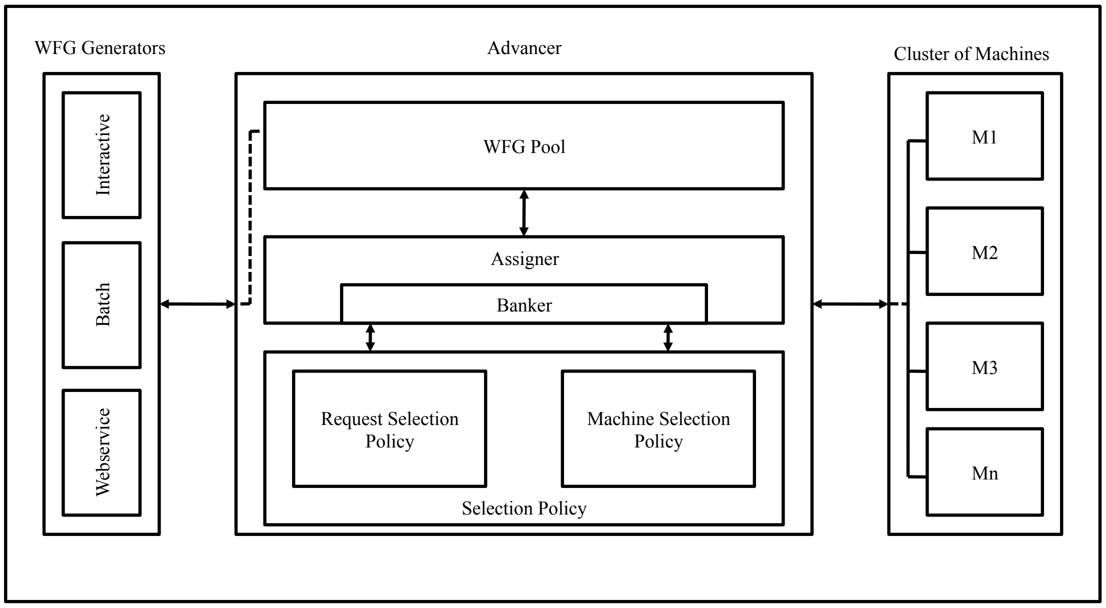


Figure 6.1: Major components of the simulation environment.

are executing. A selection policy is made of two sub-components. The first sub-component, responsible for selecting the next request to execute, is referred to as Request Selection Policy (RSP). The second sub-component, responsible for finding the most appropriate machine to execute the selected request, is referred to as the Machine Selection Policy (MSP). The four RSPs considered in this thesis are First Come First Serve (FCFS), Earliest Deadline First (EDF) [18], Least Laxity First (LLF) [19, 18], and Proportional Least Laxity First (PLLF) [14]. The MSP used in this thesis is Best Post Mapping (BPOM) [14]. Each of the main components of the scheduling framework are described in further detail in the sections to follow.

6.2 WFG Generator

The model for a WFG used in this thesis is similar to that used in [5] with one significant variation. That is, all requests forming a WFG originate from a call tree. WFG Generator uses a call tree generator to generate call trees for each WFG. The call tree generator employed by WFG Generator is probabilistic and generates different

Table 6.1: Definitions of CPU and heap memory requirements for request r.

$C_r > 0$	C_r is the total number of CPU cycles required to execute r on the fastest unloaded machine.
$D_r \geq C_r$	D_r is the ideal execution time duration of r on the fastest unloaded machine.
$U_r = C_r/D_r$	U_r is the CPU utilization factor of r.
$H_r > 0$	H_r is the maximum reachable heap memory requirement of r.
$A_r \geq H_r$	A_r is the total heap space allocated by r.
$G_r = A_r/H_r$	G_r is the garbage generation factor of r.

call trees for the three different types of WFGs. In the WFG model assumed in [14], a simple chain of requests is used in place of the call tree structures assumed here. In the model of [14] , requests do not call other requests but are themselves atomic. The inclusion of call trees in the WFG model makes scheduling and execution much more difficult because it introduces the possibility for deadlock and the necessity for an algorithm to avoid deadlock, such as the one presented in Chapter 5.

The primary function of WFG Generator in Fig. 6.1 is to provide synthetically generated WFGs to Advancer for the purpose of evaluating scheduling policies. The WFG generation process used in this thesis is probabilistic. Parameters for WFG generation rates, WFG structure, and CPU and memory requirements of the individual requests that form a WFG are defined for each WFG type. Table 6.1 summarizes the notation and definitions of basic computational and memory requirements for an individual request r. For a detailed explanation of the requirements specified in Table 6.1, refer to [5].

Each WFG has a deadline that defines the time by which all computations of a WFG should be complete. The WFG's deadline is calculated by multiplying the expected WFG duration D_w by a parameter called the deadline factor, $df \geq 1$. Low values of deadline factor indicate tight deadlines, and higher values indicates looser deadlines. The deadline of a WFG, denoted as d_w, is defined in Eq. 15.

$$d_w = df D_w. \tag{15}$$

[5]

After WFG Generator generates the interactive, webservice, or batch WFG, it sends the generated WFG to WFG Pool, which is maintained by Advancer. An interactive WFG represents a workflow generated by a client using an interactive component of the system such as a web application. These WFGs typically have a relatively small number of requests, each one having a relatively short duration and low resource requirements. Because clients demand an immediate response to interactive requests, the deadlines associated with interactive WFGs are very tight, i.e., df is close to unity. Webservice WFGs are typically more complex than interactive WFGs and have relatively more requests with longer durations, moderate resource requirements, and slightly looser deadlines. Batch WFGs are the most complicated of the three structures and typically have the most requests with the longest durations, greatest resource requirements, and loosest deadlines. Furthermore, the arrival times of batch WFGs generally have daily periodicity, which further distinguishes them from the other WFG types.

In addition to differences in arrival processes, number of requests, and deadline characteristics, the different WFG types have structural differences as well. Interactive WFGs have less opportunity for parallelization compared to webservice and batch WFGs. Batch WFGs usually have rich WFG structure with the highest chances of parallelization.

6.3 Advancer

Advancer, shown in Fig. 6.1, takes WFGs as input and assigns requests of the WFGs to machines of the cluster for execution. Advancer maintains WFG Pool to hold all

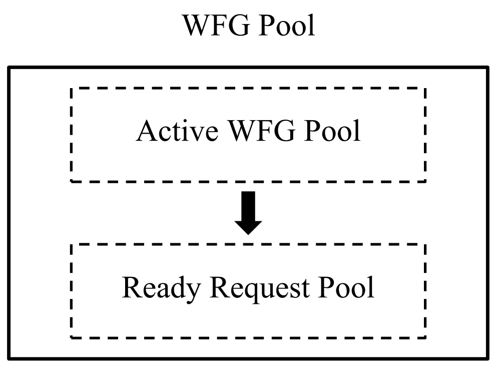


Figure 6.2: Components of WFG Pool.

the incoming WFGs. The Assigner module of Advancer is responsible for selecting requests to be considered for execution and the machines on which the selected requests should execute. The various components of Advancer are described in more detail in the subsections that follow.

6.3.1 WFG Pool

The structure WFG Pool consists of two components: Active WFG Pool and Ready Request Pool as shown in Fig. 6.2. Active WFG Pool holds all WFGs that have started but have not yet completed execution. Ready Request Pool contains requests of the WFGs that are in the "ready" state and can be considered for assignment and execution. Assigner makes use of this pool when determining which request(s) to start executing.

6.3.2 Assigner

The Assigner component of Advancer shown in Fig. 6.1 is built on the Assigner specified in [5] but has an important enhancement. This enhancement is an implementation of `Banker` described in the previous chapter. With the introduction of call trees into the WFG model, Assigner must employ some mechanism to deal with deadlock or risk infinitely tying up resources with deadlocked requests that will never finish. Still, the primary responsibility of Assigner is to select ready requests of the WFGs and assign them to machines of the cluster for execution. When making assignment decisions, Assigner can make use of computational and memory requirements assumed to be known and available for each request. As stated in Chapter 5, having access to such information is realistic in the assumed dedicated environment in which off-line profiling and/or historical logging can be performed to collect and estimate such data. Assigner can also utilize the deadline information associated which each WFG when making request scheduling decisions. Assigner employs an underlying selection policy to make decisions related to advancement of call trees in order to meet the deadlines of WFGs and consults `Banker` to ensure that these advancement decisions do not risk putting the system into a state of deadlock.

At any point during the execution of a WFG, the requests associated with one or more call trees are considered by Assigner for assignment to machines. Assigner tracks the status of each request (i.e., method instance) according to one of the following state values: blocked, ready, executing, holding, or completed. These are the same state values described earlier in Chapter 3 and shown in Fig. 3.2.

The time instant that the state of a request r transitions from blocked to ready is defined as the request's birth time and is denoted by b_r. Assigner employs a Request Selection Policy (RSP) to select the next ready request to be executed, and the Best Post Mapping [5] Machine Selection Policy (MSP) to select the best available machine for the execution. Before a selected request can be assigned to the machine selected by the MSP, Assigner must ensure that if the assignment does occur, the system cannot possibly enter a state of deadlock. To perform this task, Assigner employs an implementation of `Banker`. The implementation of `Banker` used in the simulations can have one of three results. First, `Banker` may declare that the selected request-to-machine assignment is `SAFE`. In this case, the assignment will occur and the selected request will begin executing on the selected machine. Second, `Banker`

```
Assigner($R(t_i), RSP$)
1 for scheduling instant $t_i$
2 $r_{list} \leftarrow$ RSP(R($t_i$))
3 for each $r \in r_{list}$ do
4   $M^* \leftarrow$ BestPostMapping($r$,$\mathbf{M}$)
5   if $M^* \neq \emptyset$
6     $M \leftarrow$ Banker($\mathbf{T}$,$r$,$M^*$,$\mathbf{M}$)
7     if $M \neq \emptyset$
8       assign($r$,$M$)
9     $r_{list} \leftarrow r_{list} - \{r\}$
```

Figure 6.3: Pseudocode for Assigner

may declare `SAFE` for the selected request but require that it execute on a machine other than the one selected by the MSP. In this case, the selected request will begin executing on the machine deemed suitable (i.e., `SAFE` by `Banker`). Third, `Banker` may detect that it is `UNSAFE` to execute the selected request on any machine. In the last case, the request will not be removed from the pool of requests, and the request with the next highest priority will be selected and checked for safety to begin execution on a machine.

When a request is assigned to a machine, the state of the request transitions from ready to executing. The time instant that the state of a request transitions from ready to executing is denoted as the request's start time and is denoted by s_r. The function of Assigner is to define the machine assignment and start time (s_r) for each request r.

Fig. 6.3 describes the pseudocode for Assigner. For scheduling instant t_i, Assigner uses an RSP to prioritize the collection of ready requests, which is denoted as $R(t_i)$. Associated with the collection of ready requests is a set of active call trees in the system, which is denoted as $\mathbf{T}$. RSP returns ordered ready requests, which is assigned to r_{list} (line 2). For each request r in r_{list}, Assigner uses the Best Post Mapping (BPOM) Machine Selection Policy (MSP) to determine a proposed machine, denoted as M^*, on which the execution of r should take place (line 4). When the proposed request-to-machine mapping has been determined, `Banker` is then invoked to determine if that mapping is `SAFE`. The result of `Banker` is a machine M for which `Banker` declares the assignment of r to M to be `SAFE` (line 6). M may or may not be the same machine as M^*. For most time instances in the studies conducted, M was in fact the same machine as the proposed M^*. The time instants for which M was a different machine were times where the system was under extreme load. If a `SAFE` request-to-machine mapping is determined by `Banker`, r is assigned to the machine returned by `Banker` and r begins executing on M (line 8). Whether or not a `SAFE` request-to-machine mapping is determined for r, r is removed from r_{list} and the process is repeated for each r in r_{list}.

Assigner has knowledge of machines in the cluster. Symbol $\mathbf{M}$ is used to represent the collection of machines in the cluster. A ready request can only be assigned to a machine that is declared to be available. The availability of a machine is determined using a defined threshold value associated with the machine's current efficiency. Specifically, a machine is declared to be available only if it's efficiency is above the defined threshold value. Depending on the resource requirements and threshold values, the MSP may or may not return any available machines. If it does not return any machines then Assigner excludes that request for consideration at this scheduling instant. It further considers the requests from r_{list} that may be appropriate for execution at the current time instant.

6.4 Cluster of Machines

Cluster of Machines in Fig.6.1 is modeled as a group of machines each of which follows an efficiency-based machine model. The efficiency value for a machine depends on the aggregate CPU and memory loading due to all requests executing on the machine. At each scheduling instant, the machine model provides the Assigner with updated efficiency values for all machines and also notifies Advancer of any requests that have completed execution. The machine model is described in greater detail in [5].

Chapter 7

Simulation Studies

In this chapter, the performance of the proposed deadlock avoidance Banker's algorithm is evaluated in the simulation environment along with the performance consequences it imposes on a collection of Request Selection Policies (RSPs). The exclusive use of the Best Post Mapping Machine Selection Policy (MSP) is used for all RSPs considered.

The utility of `Banker` is evaluated through simulation studies. Two important objectives of the simulation studies are to determine: (1) the efficacy of `Banker` at avoiding deadlock and (2) the impact that `Banker` has on the performance characteristics of the selection policies considered. Related to the first objective, a number of simulations are performed to verify that the use of `Banker` by `Advancer` does indeed prevent deadlock from occurring in the simulations. For the second objective, *special* simulation realizations are identified (through trial and error) for which deadlock does *not* occur *without* the use of `Banker`. For each of these realizations, the performance obtained by `SelectionPolicy` is measured both with and without the consultancy of `Banker`. Because `Banker` is conservative in making declarations of safety, `Banker` will generally declare some assignments proposed by `SelectionPolicy` to be `UNSAFE` even for a realization where deadlock does not occur without using `Banker's` declarations. It is discovered that the use of `Banker` actually enhances performance for the case study and implementations of `SelectionPolicy` considered.

7.1 Selection Policies

For the studies considered here, `SelectionPolicy` is composed of two steps. In the first step, ready requests associated with call trees are prioritized. In the second step, ready requests are considered in priority order and assigned to machines with sufficient resources to satisfy resource requirements. The priority of requests is defined according to a request selection policy (RSP). The following four RSPs are evaluated: First Come First Serve (FCFS), Earliest Deadline First (EDF), Least Laxity First (LLF), and Proportional Least Laxity First (PLLF).

The FCFS policy uses the value of the time instant that a WFG is generated to define the priority for all requests associated with the WFG. To illustrate, assume WFG A is generated before WFG B. In this case, the FCFS policy will assign a higher priority to all requests associated with WFG A (compared to the priority of all requests associated with WFG B). The FCFS policy is the simplest of the policies considered; it can make poor decisions because it does not consider the deadline of the WFGs in assigning priorities.

The EDF policy [18] prioritizes all requests of a WFG using the deadline associated with the WFG. Thus, if WFG B has a deadline that is earlier than the deadline of WFG A, the EDF policy would prioritize the execution of all requests associated with WFG B over those associated with WFG A.

The LLF policy [19, 18] prioritizes requests of a WFG according to their laxity, which is defined as the difference between the deadline of the WFG and the estimated finish time of the WFG. The rationale for giving requests with smaller values of laxity priority over larger values is because smaller values of laxity correspond to WFGs that are projected to miss their deadlines. Laxity values can be negative, and negative laxity values have priority over positive laxity values because negative laxity is an indication that the WFG deadline will likely be missed.

The PLLF policy [14] is an enhancement of the LLF policy that uses a “proportional” laxity value to prioritize ready requests of a WFG. The proportional laxity value is defined as the laxity value divided by the ideal execution of the WFG.

Unlike the FCFS and EDF scheduling policies, which assign a static priority value upon the arrival of a WFG, the priority values assigned by LLF and PLLF generally vary with time. At each simulation cycle, an estimate of each WFG’s finish time is first calculated. This estimated finish time is then subtracted from the WFG’s deadline, which yields the WFG’s laxity value at that time instant. These and other RSPs are described in further detail in [14].

7.2 Expiremental Setup

In this thesis, a WFG generation scenario is considered representing a 24 hour period. Further detail for the experimental setup is given in the sections below.

7.2.1 Case Study

A case study is considered in which a single day is divided into three consecutive epochs. These three epochs are associated with WFG generation characteristics for a typical operational business day. The first epoch is from time = 0 to time = 11 hours; the second epoch is from time = 11 to time = 12 hours; and the third epoch is from time = 12 to time = 24 hours (refer to Fig. 7.1). During the first and third epochs, only Interactive and Webservice WFGs are generated. During the second epoch, all three types of WFGs are generated. The first and third epochs represent periods of time before and after a relatively short epoch in which Batch WFGs arrive. The start and end times of the second epoch are defined by terms of service-level agreements

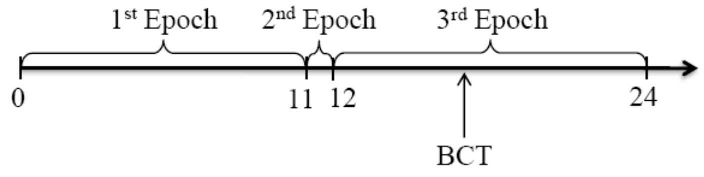


Figure 7.1: Time-line illustrating the three epochs.

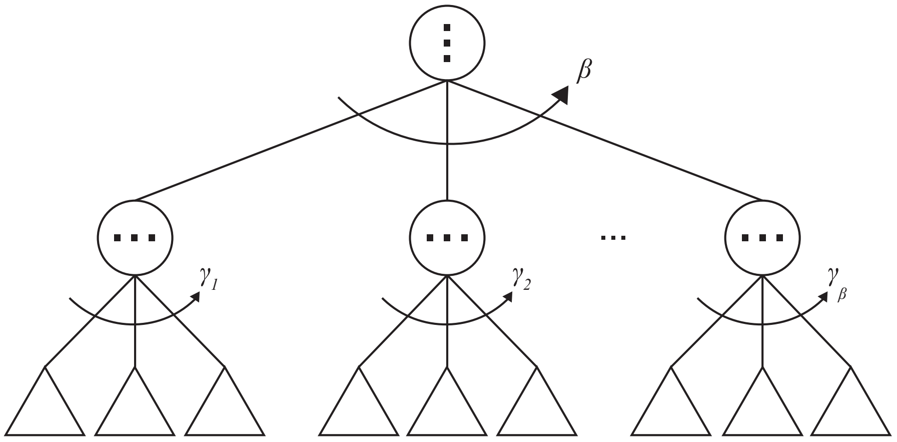


Figure 7.2: WFG structure used in simulations.

(SLAs) related to timing of Batch WFG submission and execution. Typical terms of SLAs specify that daily Batch WFGs submitted within a specified time period will be completed by an agreed upon deadline.

Fig. 7.2 shows the structure of the two-level WFGs assumed in the simulations. In this figure, β is the number of control nodes that are direct children of the root and are assumed to be sequentially executed. Note that the structure of the WFG used in the simulation studies (Fig. 7.2) is similar to the example WFG associated with the BlueBox SOA discussed in Chapter 4 (see Fig. 4.2). The notation γ_i represents the number of call trees associated with the i^{th} parallel control node. The call trees of each parallel control node are assumed to be independent and may be executed concurrently. The values for β and γ_i vary for the three different WFG types. The parameter value ranges and distributions associated with the simulation studies are summarized in Table 7.1. The parallelization factor is needed in determining a base

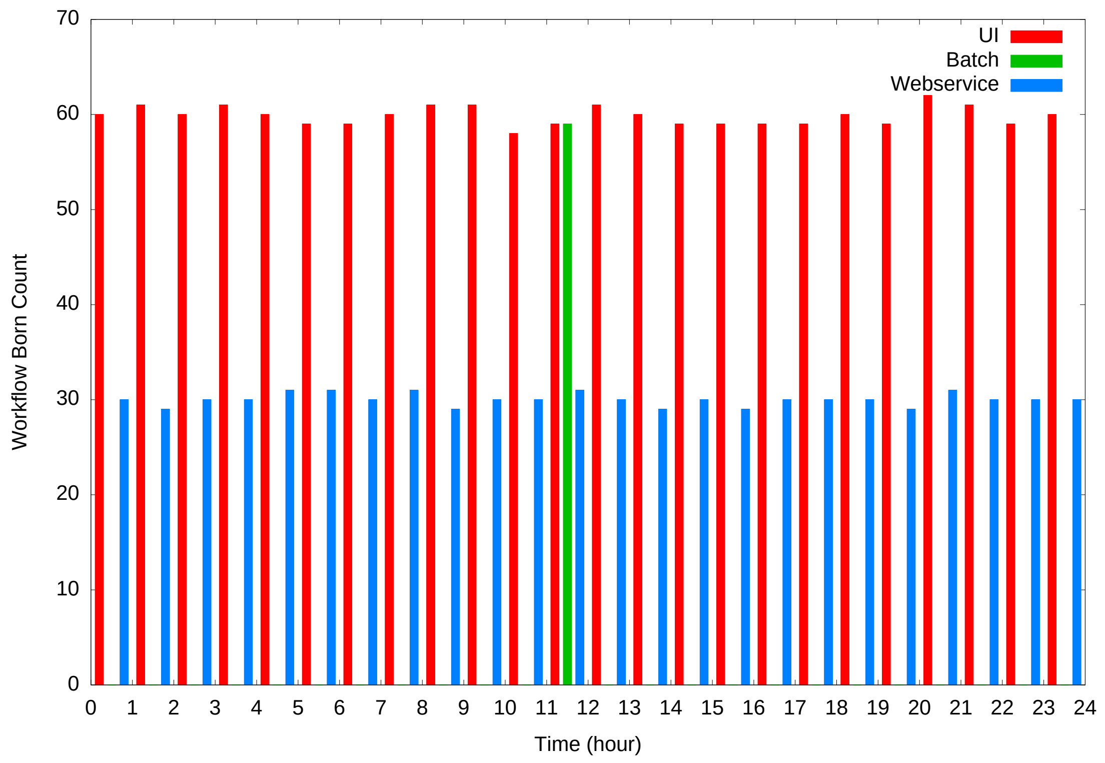


Figure 7.3: Arrival count realization for Case Study.

deadline for each generated WFG; it defines the degree of parallelism assumed for executing parallel call trees associated with a common control node. Once a base deadline is determined for a WFG, it is multiplied by the Deadline Factor (last row in the table) to define the deadline for the synthetically generated WFG.

Table 7.1: WFG parameter value ranges used for case study. Values in [Min, Max] sampled from uniform distribution.

Parameter	**Interactive WFG**	**Webservice WFG**	**Batch WFG**
Inter-Arrival Time (secs)	60	120	60
β	[1, 1]	[1, 3]	[3, 5]
γ_i	[1, 2]	[2, 3]	[5, 20]
Parallelization Factor	2	2	2
WFG Deadline Factor	[1.1, 1.2]	[1.3, 1.5]	[1.3, 1.5]

Interarrival times of the Interactive and Webservice WFGs in the Case Study are

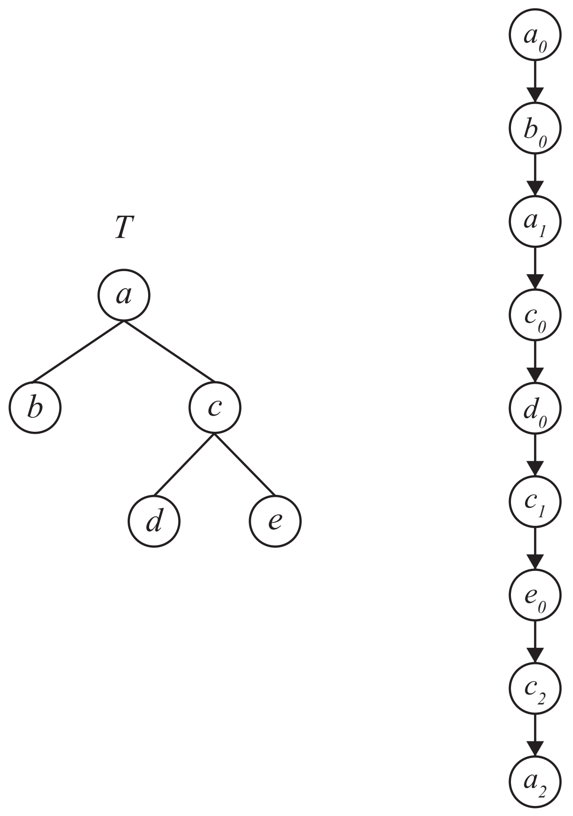


Figure 7.4: Chain of segments from the expansion of call tree T.

60 seconds and 120 seconds respectively. The interarrival time of the Batch WFGs is assumed to be 60 seconds during the second time epoch from hour 11 to hour 12; Batch WFGs do not arrive outside this one-hour interval. All interarrival times are generated using a Poisson process. An example arrival count realization of the arrivals of WFGs into the system are shown in Fig 7.3.

A chain of method segments is formed by performing a depth first traversal of a call tree. For an illustrative example, consider Fig. 7.4. The call tree on the left is expanded into the chain of segments on the right that represents "requests" with precedence constraints. For the simulations conducted, three call tree stuctures were considered for each WFG type. The structures for Interactive and Webservice WFGs are shown in Fig. 7.5. Likewise, Fig. 7.6 shows the structures for Batch WFGs.

Requests belonging to a call tree are assumed to have certain CPU and memory

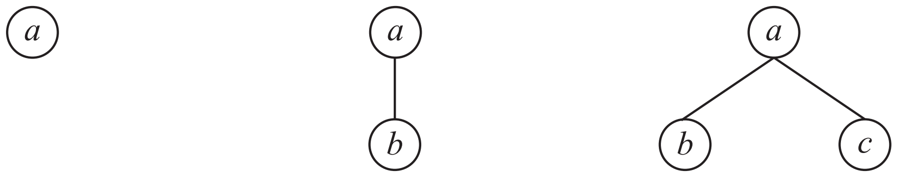


Figure 7.5: Call Tree structures for Interactive and Webservice WFG types.

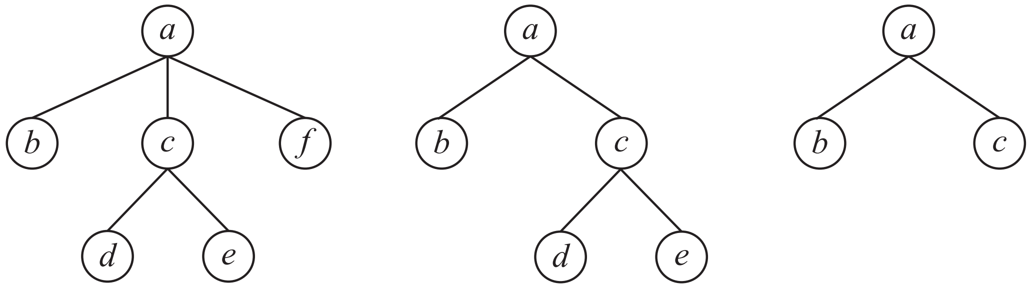


Figure 7.6: Call Tree structures for Batch WFG types.

characteristics associated with the specific WFG type to which they belong. The CPU and memory characteristics for the three WFG types are summarized in Table 7.2.

Table 7.2: Parameter value ranges for requests associated with call trees of Figs. 7.5 and 7.6.

Request Parameter	**Interactive WFG**	**Webservice WFG**	**Batch WFG**
Ideal Duration (secs)	[1.0, 2.5]	[10.0, 50.0]	[50.0, 175.0]
CPU Utilization	[0.5, 1.0]	[0.5, 1.0]	[0.5, 1.0]
Memory Usage	[0.05, 0.15]	[0.05, 0.10]	[0.05, 0.10]

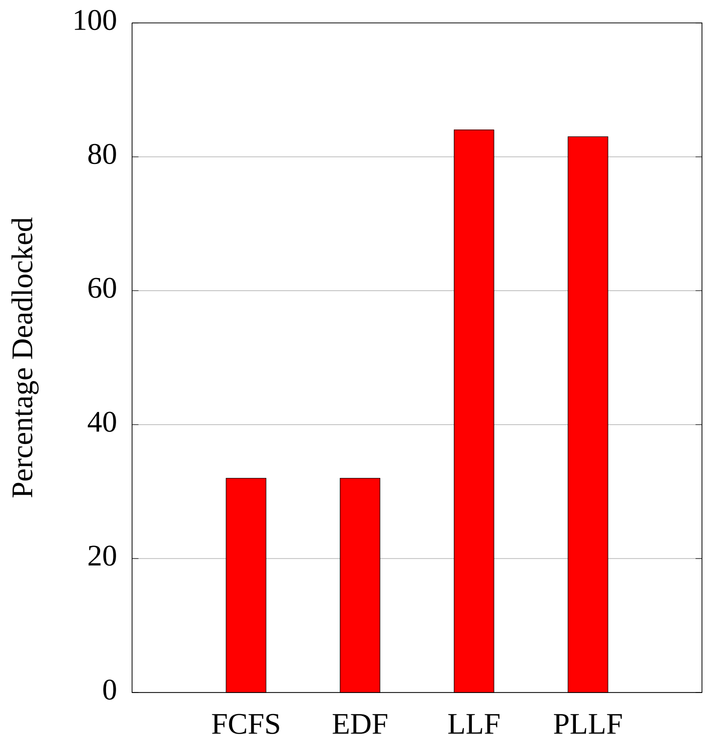


Figure 7.7: RSPs' likelihood to deadlock without utilizing Banker.

7.3 Deadlock Avoidance Results

For the case study described in the previous subsection, one hundred simulations were conducted with the four RSPs described in Section 7.1. Each of the RSPs have different characteristics leading to different likelihoods for experiencing deadlock. Assuming `Advancer` does not utilize `Banker`'s declaration of safety, the observed statistics for the occurrence of deadlock for each RSP are summarized in Fig. 7.7. A second set of simulations was conducted in which `Banker`'s declaration of safety was utilized. For this set of simulations, deadlock was avoided for all RSPs.

As an illustration of how `Banker` interacts with the `SelectionPolicy` to avoid deadlock, consider Fig. 7.8. The figure shows the number of active call trees in the system for two simulations conducted, both employing PLLF as the RSP. In one simulation, `Advancer` utilizes `Banker`'s declarations of safety, and in the other it does not. From the figure, it is apparent that when the system is underloaded

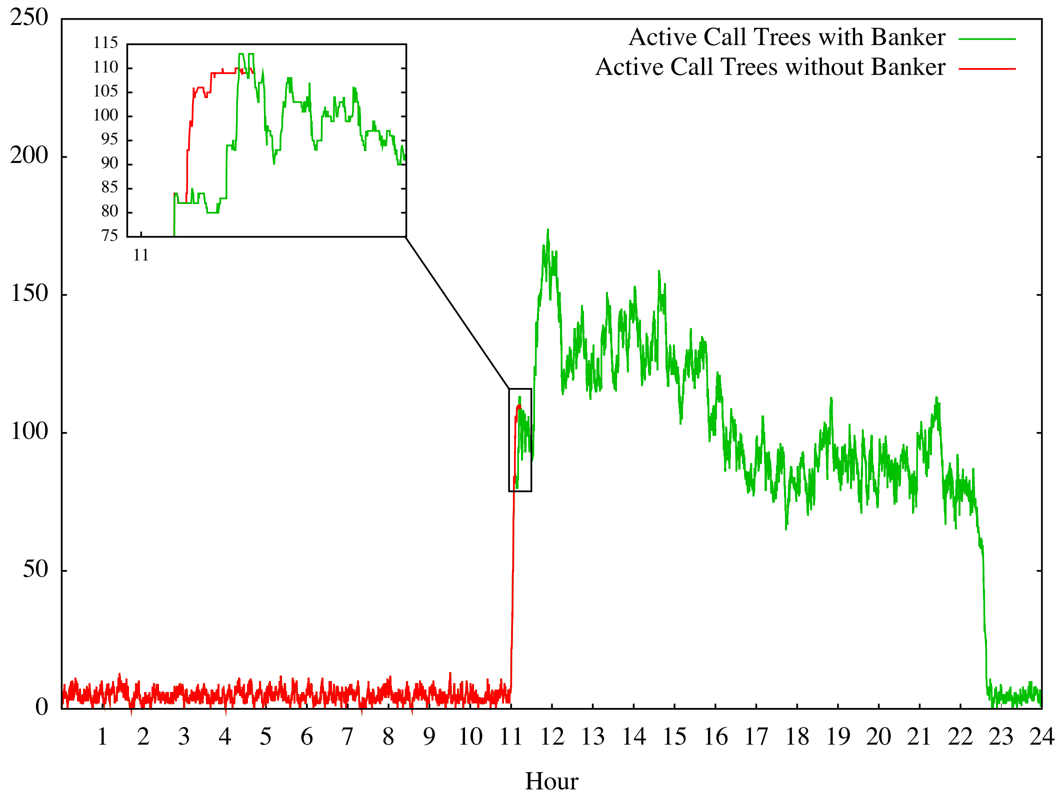


Figure 7.8: Active call trees with and without Banker for a select simulation.

(hour one to hour eleven) `Banker` does not interfere after the scheduling decisions of `SelectionPolicy`. This is because the system is under-loaded and any decision made by `SelectionPolicy` will leave the system in a `SAFE` state. However, as resources become scarce, `Banker` begins to force `Assigner` to use alternate `SelectionPolicy` decisions to ensure the system is left in a `SAFE` state. Just after hour eleven, large quantities of batch WFGs start arriving. When this occurs, more and more call trees become active. As illustrated by the inset portion of the figure, if decisions of `SelectionPolicy` go unchecked, deadlock occurs. In the simulation without `Banker`, deadlock is observed at the time instant where the graph ends, just after hour eleven. Also from inset portion of the figure, it is apparent precisely when `Banker` forces some active call trees to finish, thereby relieving memory pressure, before allowing more call trees to begin executing. `Banker` delays the start/execution of certain call trees in order to ensure sufficient resources necessary for active call trees to complete.

Because `Banker` sometimes forces `SelectionPolicy` to make decisions different than it would if it was running standalone, it follows that `Banker` has influence on the performance of `SelectionPolicy` in terms of tardiness relative to workflow deadlines. Interestingly, `Banker` does not negatively affect the performance of `SelectionPolicy` in terms of tardiness for the simulations considered. In fact, in most cases considered, `Banker` actually improves the performance of `SelectionPolicy`. In the sections to follow, the effects of `Banker's` are evaluated for each of the RSPs considered in this thesis.

7.4 Banker's Effect on FCFS

Compared to the other policies evaluated for the case study considered here, FCFS without `Banker` performs the poorest in terms of the percentage of workflows tardy. This is not a surprising result and similar results were realized in [5] for WFGs comprised of atomic requests. One possible explanation for the poor performance is that as the system becomes overloaded with the arrival of batch workflows around hour eleven, the system becomes congested and eventually devotes all available resources to executing the batch workflows. In the meantime, all arriving workflows are queued behind long-running batch workflows, regardless of deadline or duration, and are not executed until adequate system resources are no longer consumed by the batch workflows. Fig. 7.9 illustrates the effect `Banker` has on FCFS for the case study considered. Positive values of normalized tardiness indicate a WFG completed after its deadline; negative values indicate a WFG finished before its deadline. Interestingly, when FCFS employs `Banker` the percentage of tardy workflows decreases dramatically. This occurs, at least in part, because `Banker`, while ensuring deadlock does not occur, does not allow the system to devote all resources to servicing the resource-intensive batch workflows. As a side effect, enough system resources

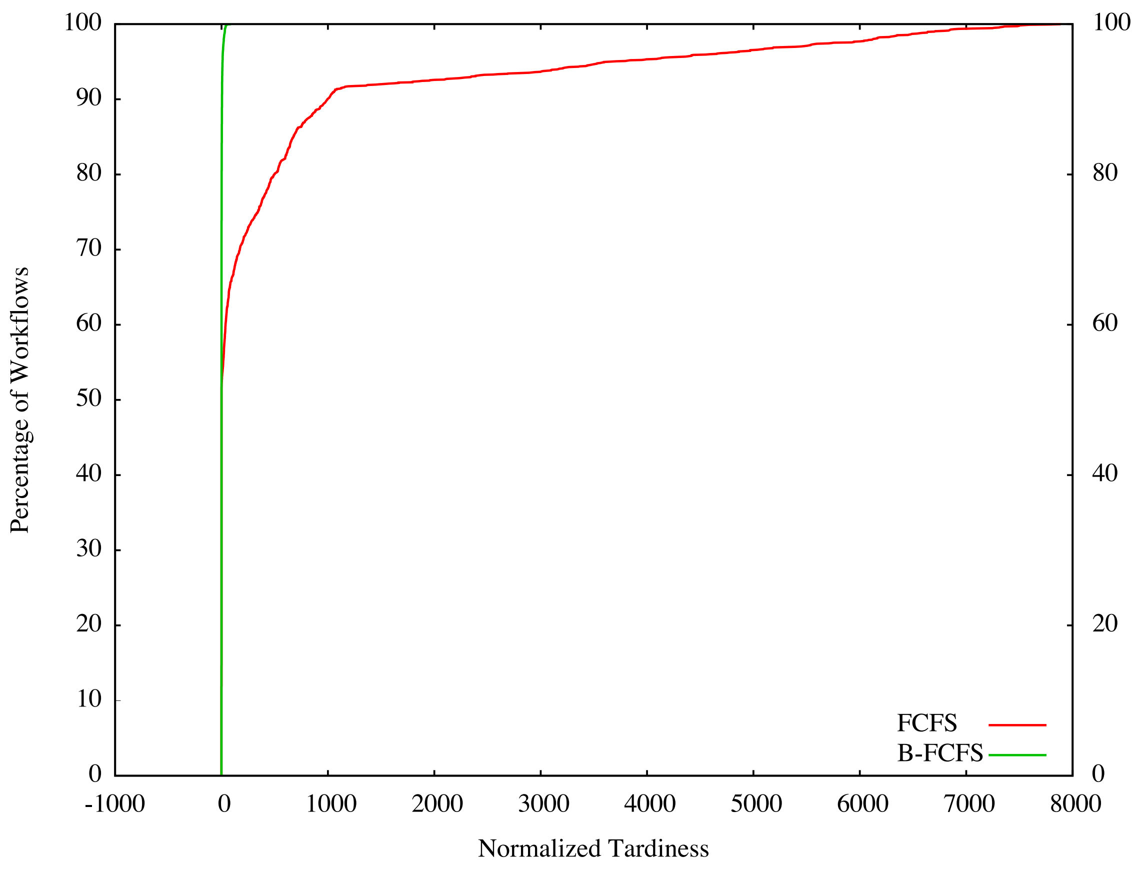


Figure 7.9: Percentage of workflows as a function of normalized tardiness for FCFS and FCFS utilizing Banker (B-FCFS) for a select simulation.

are available to service the smaller interactive and webserivce workflows that arrive during the time of heavy load. The same performance data of Fig. 7.9 is provided in Fig. 7.10 on a tighter normalized tardiness scale.

Fig. 7.11 illustrates how `Banker` influences the number of "active" call trees in the system. A call tree T is said to be "active" from the time instant the first request of T begins execution until the time instant the last request in T is completed. In this particular case, the simulation did not deadlock when `Banker` was not employed. Of particular importance is the portion of the graph corresponding to hours 0-11. During this time, the load on the system is relatively light and `Banker` does not influence the system to do anything different than the FCFS dictates. In this way, `Banker` is a passive algorithm and does not interfere with the decisions of the underlying

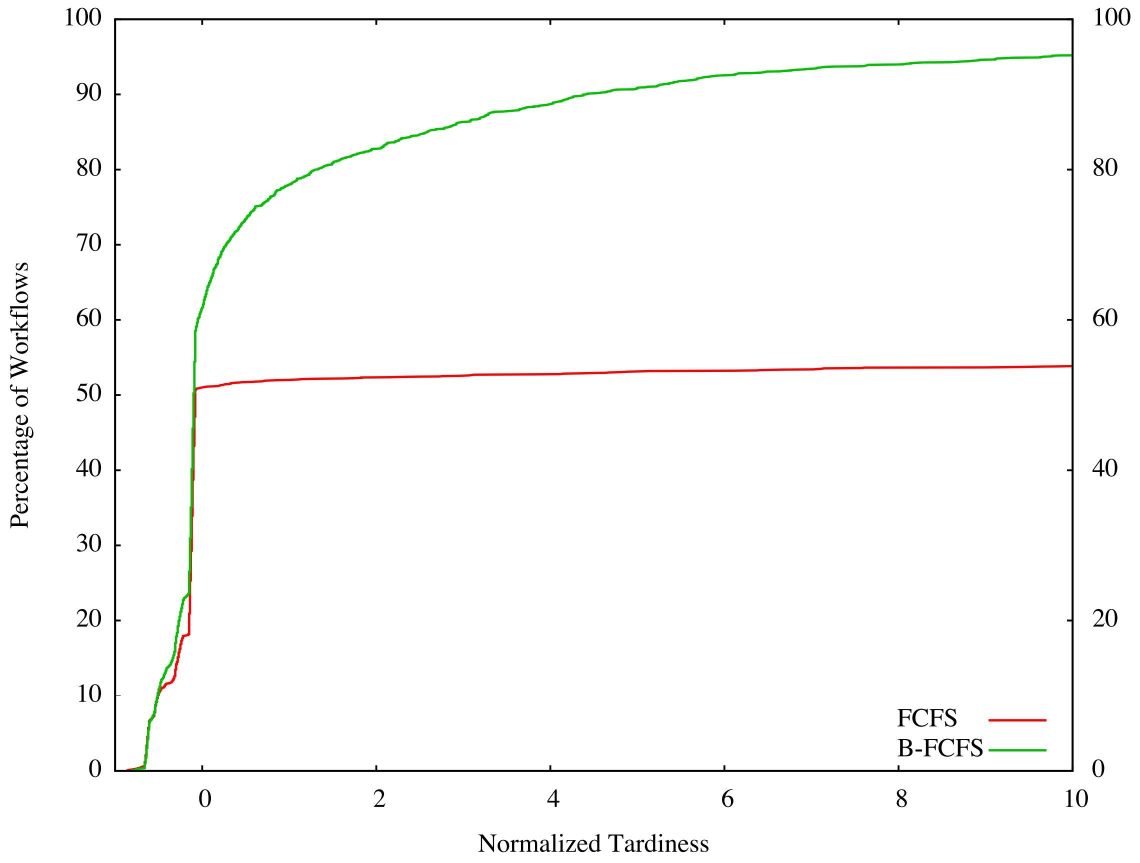


Figure 7.10: Percentage of workflows as a function of normalized tardiness for FCFS and FCFS utilizing Banker (B-FCFS) for the same simulation of Fig. 7.9 on a normalized tardiness scale from -1 to 10.

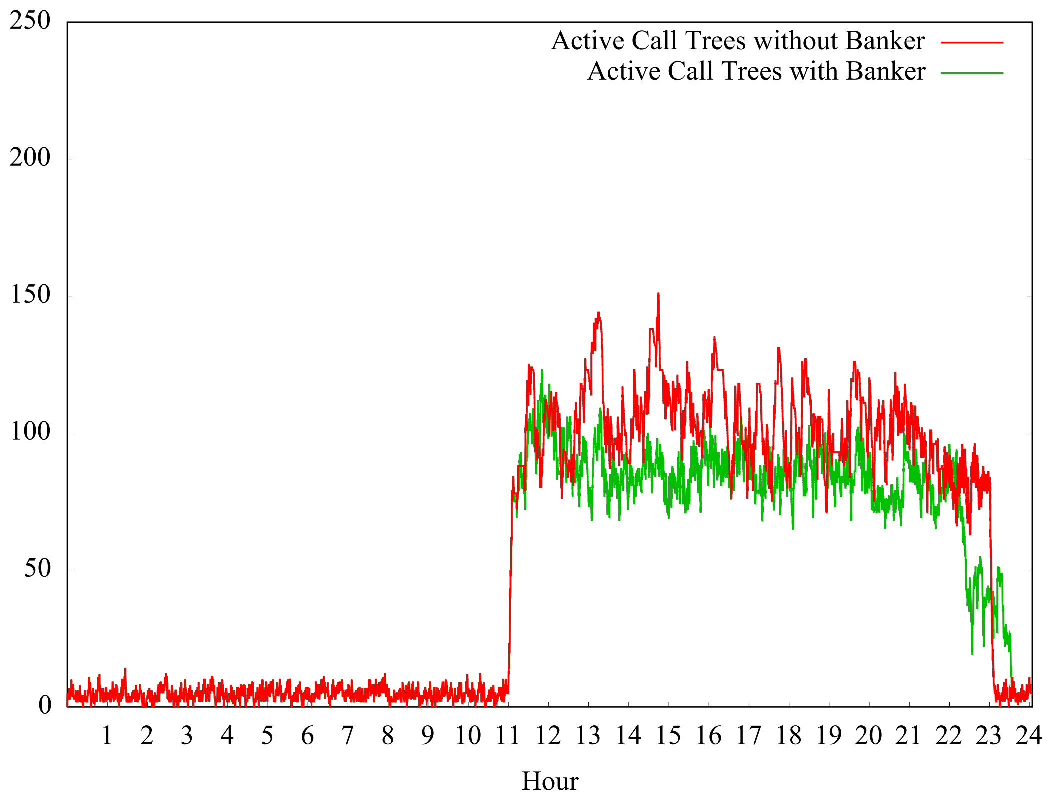


Figure 7.11: Active Call Trees with and without Banker for a select simulation employing the FCFS RSP.

scheduling policy unless it is necessary to avoid deadlock. According to the figure, the number of active call trees with `Banker` in place is not dramatically different than the number of active call trees without `Banker`. One explanation for this is that when `Banker` delays a long-running, resource-intensive call tree, it allows another smaller less resource-intensive call tree to execute in its place. However, it is important to note that the use of `Banker` does generally reduce the number of "active" call trees, thus reducing resource utilization.

7.5 Banker's Effect on EDF

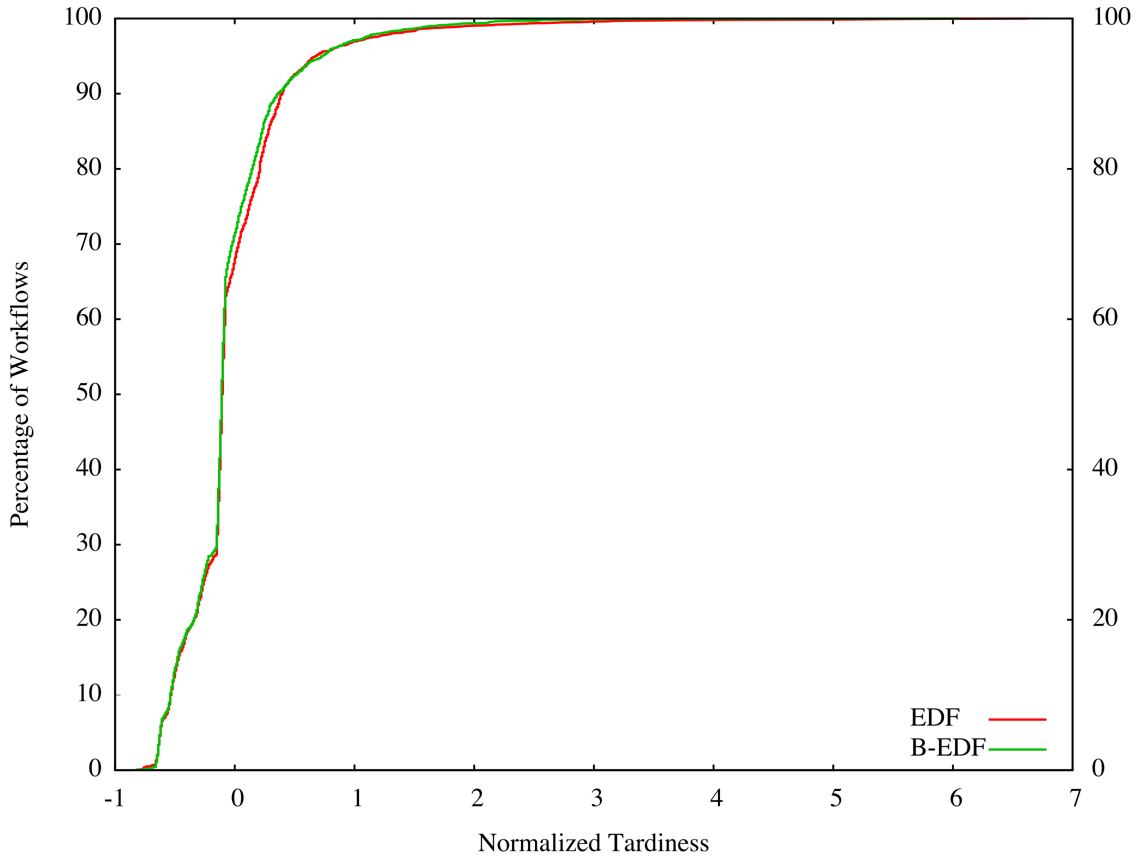


Figure 7.12: Percentage of workflows as a function of normalized tardiness for EDF and EDF utilizing Banker (B-EDF) for a select simulation.

For the case study considered, EDF outperformed every other policy running

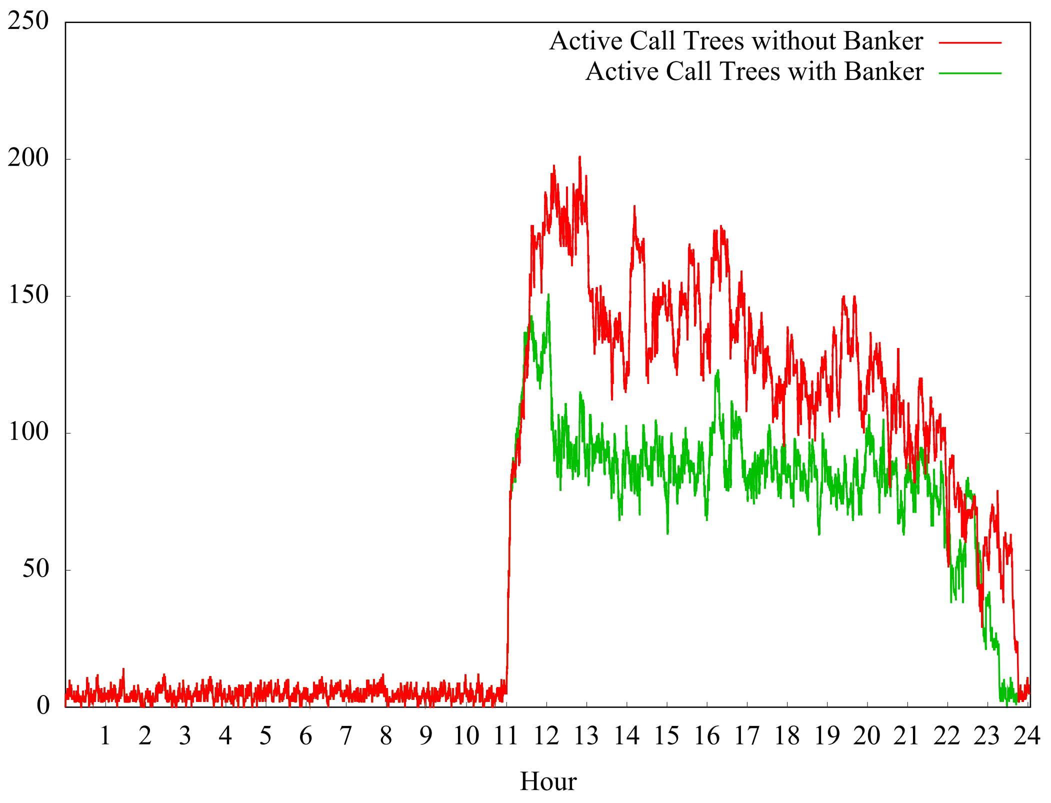


Figure 7.13: Active Call Trees with and without Banker for a select simulation employing the EDF RSP.

without `Banker`. Fig. 7.12 shows the effects of `Banker` in terms of percentage of tardy workflows. Overall, `Banker` does not seem to either improve or inhibit the performance of EDF for the simulations conducted.

Fig. 7.13 shows the active call trees for a simulation conducted with and without `Banker`. According to the figure, `Banker` reduces the number of active call trees in the system for most times throughout the duration of the simulation.

7.6 Banker's Effect on LLF

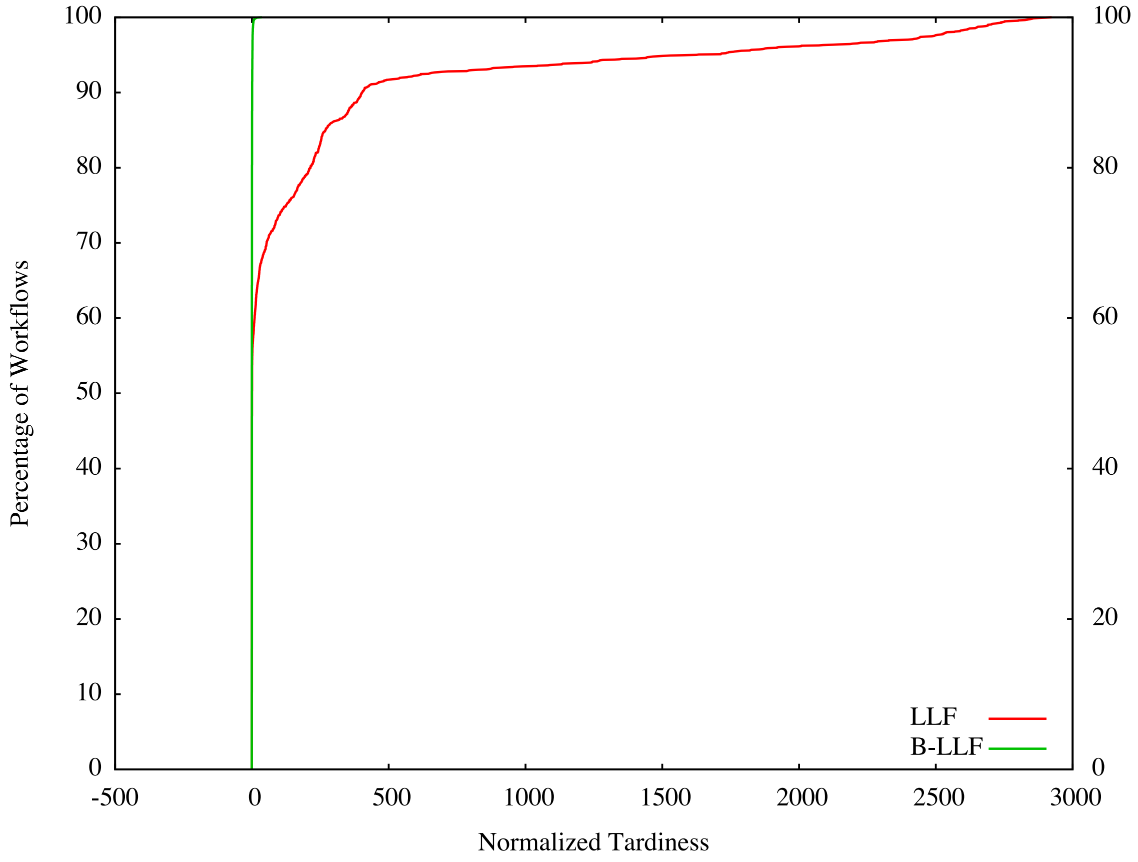


Figure 7.14: Percentage of workflows as a function of normalized tardiness for LLF and LLF utilizing Banker (B-LLF) for a select simulation.

Recall from Fig. 7.7 that LLF exhibited the highest likelihood for deadlock of all the RSPs considered. In addition, LLF without `Banker` did not perform particularly

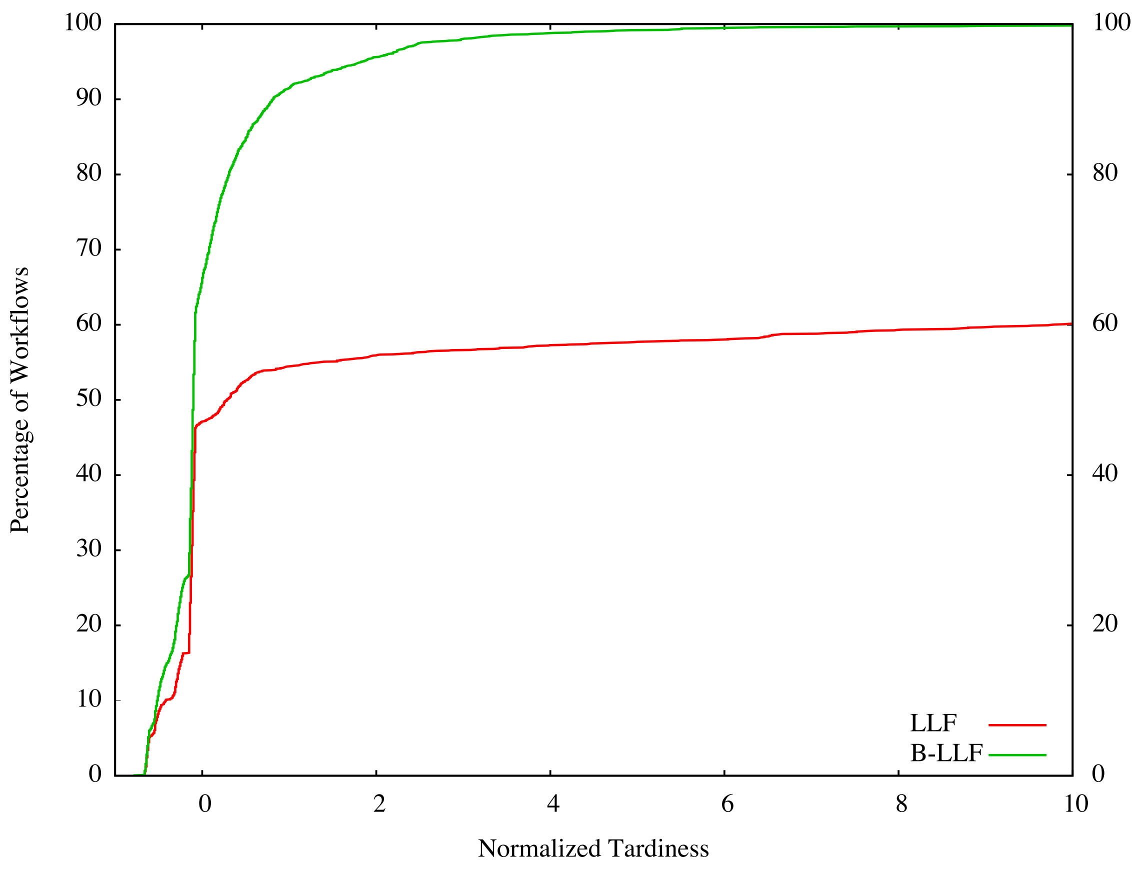


Figure 7.15: Percentage of workflows as a function of normalized tardiness for LLF and LLF utilizing Banker for a select simulation on a normalized tardiness scale from -1 to 10.

well in terms of workflow tardiness. In fact, only FCFS performed worse. However, when `Banker` is used in conjunction with LLF, the performance of LLF in terms of workflow tardiness improves drastically. Fig. 7.14 and Fig. 7.15 illustrates this.

Intrinsic to LLF is a tendency to context switch between executing workflows. As a result of this tendency, LLF tends to leave many active call trees in the holding state. Fig. 7.16 illustrates this characteristic and also the substantial difference `Banker` makes when coupled with LLF. One explanation for this large difference is that `Banker` forces LLF to finish active call trees before starting new ones. Intuitively, the higher the number of active call trees in the system, the more system resources are consumed in general. This simulation provides yet another case where `Banker`

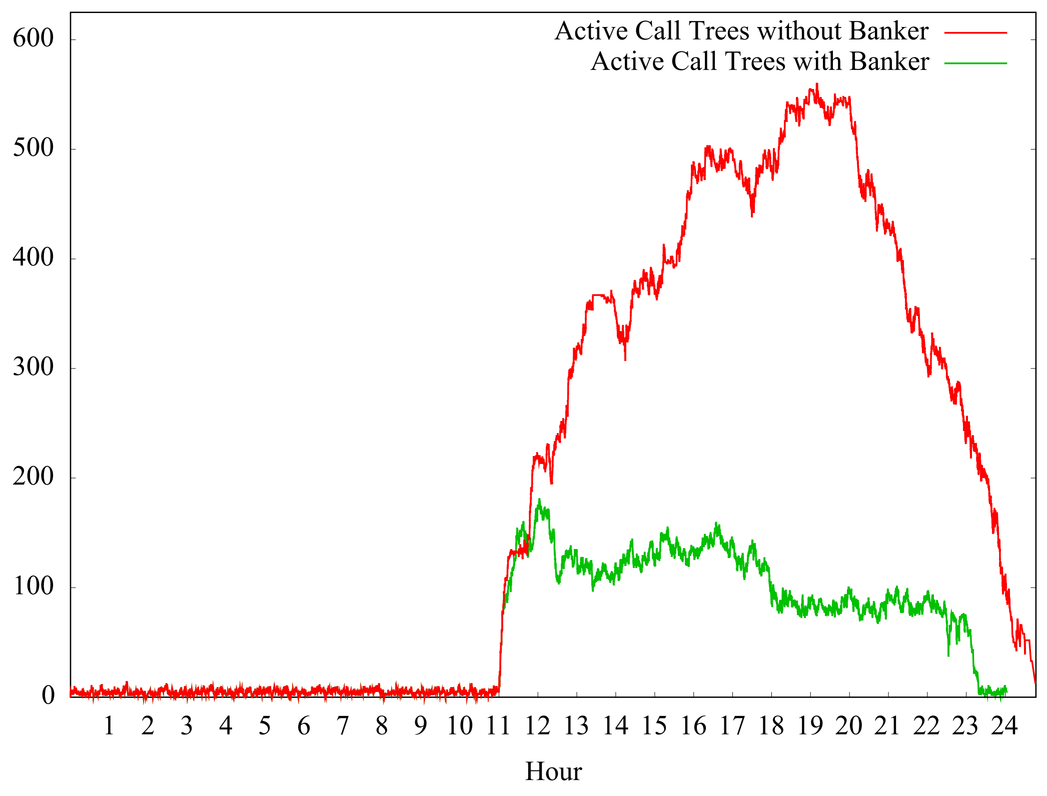


Figure 7.16: Active Call Trees with and without Banker for a select simulation employing the LLF RSP.

drastically improves the performance of the RSP.

7.7 Banker's Effect on PLLF

Just as it was determined in [14], PLLF outperformed LLF in terms of workflow tardiness for the case study considered. In fact, PLLF performed quite well in this category for the case study considered here. Only EDF showed better performance without `Banker`. However, Fig. 7.17 shows that `Banker` still improved PLLF performance in terms of workflow tardiness for this select simulation.

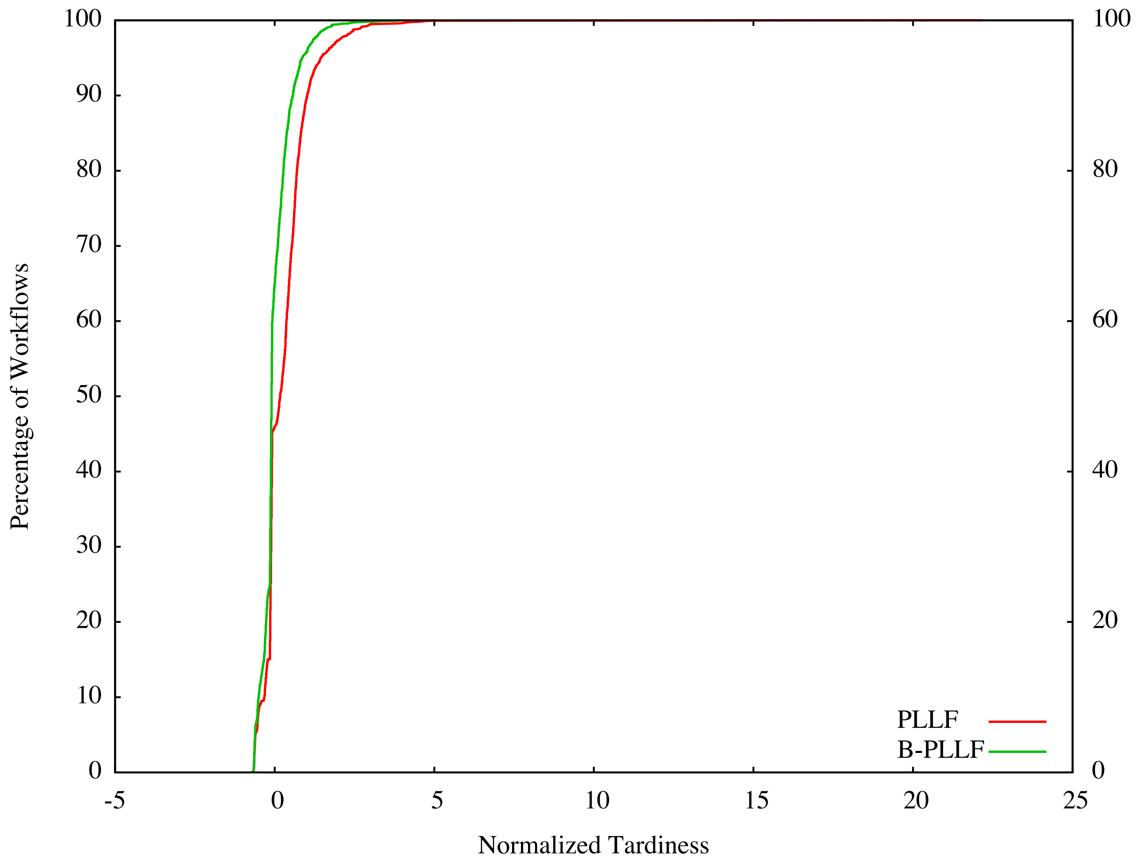


Figure 7.17: Percentage of workflows as a function of normalized tardiness for PLLF and PLLF utilizing Banker (B-PLLF) for a select simulation.

Though not quite as poor as LLF, PLLF exhibits the same intrinsic tendency to context switch between executing workflows. Again, this tendency leads to an

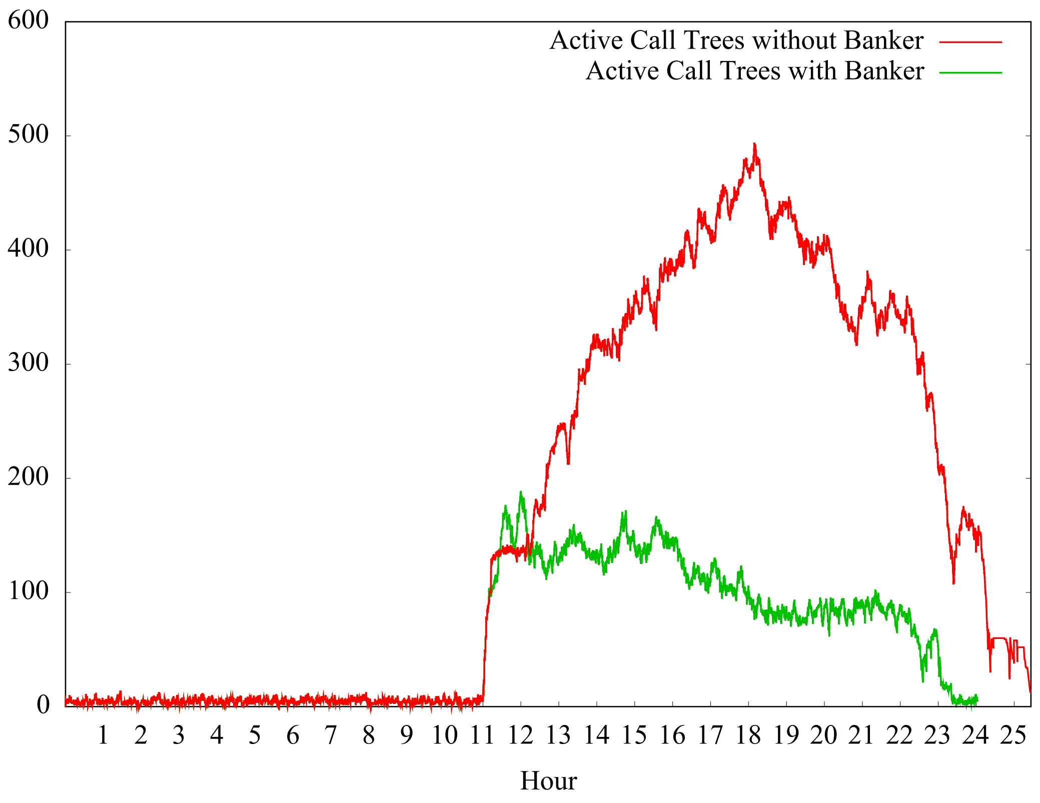


Figure 7.18: Active Call Trees with and without Banker for a select simulation employing the PLLF RSP.

increased number of active call trees in the system. Fig. 7.18 shows the active call trees for a select simulation conducted with and without `Banker`. From the figure, it is quite obvious that `Banker` makes very different decisions in terms of what call trees get started and finished. The end result, however, is a more efficient and better performing overall `SelectionPolicy`.

Chapter 8

Conclusions

A new deadlock avoidance algorithm, derived from Dijkstra's Bankers Algorithm [11], is introduced for avoiding deadlock on a distributed SOA. The algorithm works by observing the current state of the resources in the system along with a worst case estimate of future resource requirements and permitting the execution of only those call trees that will keep the system in a safe state. Through simulation studies, it is observed that the algorithm is "non-intrusive" when the system resources are plentiful and does not influence the decisions of the underlying selection policy.

The simulation results indicate that using `Banker` has multiple benefits. The selection policies considered in the simulations: FCFS, EDF, LLF, and PLLF have different characteristics leading to different likelihoods for experiencing deadlock. However, when `Banker` is utilized in conjunction with any of the selection policies, deadlock is avoided. In addition to ensuring deadlock avoidance, `Banker` can influence `SelectionPolicy` to make decisions that improve the performance of the system in terms of workflow tardiness. One hypothesis is that the performance benefits observed when utilizing `Banker` are attributed to `Banker` forcing `SelectionPolicy` to concurrently execute different collections of call trees, prefering to run a larger number of shallow and/or undemanding call trees as opposed to a fewer number of deeply nested and/or resource intensive call trees.

Future work is planned in two areas. The first of which is the integration of the deadlock avoidance algorithm developed in this thesis into the BlueBox SOA. Another area of future work is the application of the algorithm in an environment where task migration is possible.

Bibliography

[1] Albert Y. H. Zomaya, editor. *Parallel and distributed computing handbook.* McGraw-Hill, Inc., New York, NY, USA, 1996.

[2] Selim G. Akl. *Parallel computation: models and methods.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[3] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[5] Hira Kaji Shrestha. Scheduling workflows on a cluster of memory managed multicore machines. Master's thesis, University of Oklahoma, Norman, Oklahoma, December 2009.

[6] Mike P. Papazoglou and Willem-Jan Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16(3):389–415, 2007.

[7] Nilo Mitra and Yves Lafon. SOAP version 1.2 part 0: Primer (second edition). Technical report, W3C, April 2007. http://www.w3.org/TR/2007/REC-soap12-part0-20070427/.

[8] Mark Hapner, Rahul Sharma, Rich Burridge, Joseph Fialli, and Kim Haase. *Java Message Service API tutorial and reference: messaging for the J2EE platform.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[9] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[10] John O'Hara. Toward a commodity enterprise middleware. *Queue*, 5(4):48–55, 2007.

[11] W. Dijkstra, Edsger. *Selected Writings on Computing: A Personal Perspective.* Springer Verlag, 1982.

[12] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, 1971.

[13] Abraham Silberschatz. *Operating System Concepts.* John Wiley & Sons, Inc., New York, NY, USA, 2007.

[14] H. K. Shrestha, N. Grounds, J. Madden, M. Martin, J. K. Antonio, J. Sachs, J. Zuech, and C. Sanchez. Scheduling workflows on a cluster of memory managed multicore machines. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '09)*, July 2009.

[15] Shau-Dong Lang. An extended banker's algorithm for deadlock avoidance. *IEEE Trans. Softw. Eng.*, 25(3):428–432, 1999.

[16] César Sánchez, Henny B. Sipma, Zohar Manna, and Christopher D. Gill. Efficient distributed deadlock avoidance with liveness guarantees. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 12–20, New York, NY, USA, 2006. ACM.

[17] J. Madden, N. G. Grounds, J. Sachs, and J. K. Antonio. The gozer workflow system. *15th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2010), cosponsors: IEEE Computer Society and ACM, in, Proceedings of the 24th International Parallel and Distributed Processing Symposium (IPDPS 2010)*, April 2010.

[18] V. Salmani, M. Naghibzadeh, A. Habibi, and H. Deldari. Quantitative comparison of job-level dynamic scheduling policies in parallel real-time systems. *Proceedings TENCON, 2006 IEEE Region 10 Conference*, November 2006.

[19] S. H. Oh and S. M. Yang. A modified least-laxity-first scheduling algorithm for real-time tasks. *Proceedings of the 5th International Workshop on Real-Time Computing Systems and Applications (RTCSA '98)*, pages 31–36, October 1998.

Acronyms

SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
WSDL	Web Service Definition Language
RSP	Request Selection Policy
MSP	Machine Selection Policy
FCFS	First Come First Serve
LLF	Least Laxity First
PLLF	Proportional Least Laxity First
EDF	Earliest Deadline First
BPOM	Best Post-Mapping
SLA	Service Level Agreements
CORBA	Common Object Request Broker Architecture
JMS	Java Message Service
AMQP	Advanced Message Queuing Protocol