# A Real-Time Distributed Resource-Granting Task Scheduler for a Virtual Infrastructure Management System

by

Yunlu Yang

S.B., Massachusetts Institute of Technology (2011)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2012

Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
June 1, 2012

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Frans Kaashoek, Professor EECS, Thesis Supervisor
June 1, 2012

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Dr. Min Cai, Staff Engineer, VMware Inc., Thesis Co-Supervisor
June 1, 2012

Accepted by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Prof. Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

# A Real-Time Distributed Resource-Granting Task Scheduler for a Virtual Infrastructure Management System

by

Yunlu Yang

Submitted to the Department of Electrical Engineering and Computer Science
on June 1, 2012, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

## Abstract

This M.Eng. thesis introduces a distributed real-time task scheduler that is designed and implemented to replace the role of a lock server for a cloud service that manages data centers that contain virtual machines. A round-robin scheduling algorithm is proposed that orders tasks based on their needs of locks, serializing tasks that conflict in their lock use. Tasks are required to obtain all locks before execution. The distributed part of the scheduler is based on Zookeeper, an open-source data service for distributed applications. The scheduler includes a failover scheme that relies on task states stored on Zookeeper. The design is highly available—the scheduler will keep scheduling tasks as long as at least one instance is running. Experimental results show that the scheduler cuts down overall task completion time by approximately a half compared to a lock server on both benchmarks used.

# Acknowledgments

I would like to thank my mentor at VMware, Min Cai, for helping me all the way through the thesis project since the summer of 2010. He could always inspire me to do more and he always managed to give me new ideas when I felt stuck. I would like to thank John Cho, for his great effort in initiating the VIMS prototype and inspiring me to join, and for his patience in explaining the details and getting me familiar with the system. I thank my manager, Haripriya Rajagopal, for always giving me honest feedback. I thank many other colleagues at VMware for their invaluable suggestions, comments and assistance. They have also helped me with many VMware tools and environmental setups.

I would like to thank my supervisor, Frans Kaashoek, for helping me define and finalize my thesis. I have benefited from his sharp insights and I appreciate his detailed attention to my arguments. The thesis would never be as complete without him.

I would like to thank my parents for their unconditional love and support throughout my undergraduate and M.Eng. years at MIT. I would like to thank my cousins for always being strong advocates for me, and my friends who have always listened when I needed to talk.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

In a non-distributed program where multiple threads run on a single machine, access to shared data (reads or updates) are commonly protected by in-memory locks. Similarly, for a distributed system where processes coordinate over the network, access to shared data also needs to be properly synchronized to protect against data corruption or inconsistent system states. Shared data in a distributed system usually refers to data stored in a storage device or cache that is accessed by multiple processes, such as a shared database or distributed in-memory cache. The common approach to data synchronization in distributed systems is to use a lock server that all processes go through to acquire locks that protect the corresponding data before performing reads or updates on the data. Lock servers usually implement a primary-backup scheme to ensure availability.

In this thesis, the idea of using a task scheduler instead of a lock server is explored for synchronizing operations of a multi-server application that serves remote client requests to perform operations such as powering on a virtual machine or adding a new host in the cloud. This multi-server application is called the virtual infrastructure management system (VIMS).

## 1.1 Motivation

The thesis idea was formed in the process of developing a new prototype of the distributed VIMS. The new prototype is a multi-server application where operations such as creating or removing a new virtual machine, powering on or off an existing virtual machine, shutting down a host machine, moving a virtual machine to a different host, etc., are performed by client sending the appropriate remote procedure calls to one of the server instances. VIMS used to be a single-machine design that uses in-memory locks, which does not scale well. As the hierarchical locking scheme grew in complexity, deadlock problems became more popular and concurrency suffered from inappropriate choices of locking levels. A task scheduler was proposed in order to eliminate deadlocks and improve concurrency. This thesis explored this proposal.

### 1.1.1 Synchronization on Data Access Is Needed

Two client calls to the API service can conflict if they affect the same virtual infrastructure components. As a simple example, two powering calls for the same virtual machine (`VirtualMachine.powerOn` and `VirtualMachine.powerOff`) cannot be executed at the same time. The states of the virtual components need to be protected from concurrent modifications.

### 1.1.2 Limitations on Using Locks

A single-machine system can use in-memory locks to protect data or states that are accessed by multiple threads. Similarly, for a distributed system where multiple machines coordinate over the network, shared states can be protected by a commonly accessible lock server. Servers acquire and release locks in the same fashion; the only difference is that the communication happens over the network, not in local memory.

VIMS used to be a single-machine design and uses hierarchical in-memory locks that roughly correspond to the organization of the cloud infrastructure. I.e. top-level locks protect the whole infrastructure (or data center); each level below it protects a subcomponent contained by its parent-level components. For example, data center contains clusters, which contain hosts. So locks for a cluster is one level below data center locks, and host locks are one level below cluster locks. Threads are required to acquire locks in order, from the highest-level locks to the lowest-level locks.

As the system grows more complex over the years, the hierarchical locking scheme has begun to show its limitations:

- The complexity of the locking hierarchy grows and deadlock problems become more prominent and also harder to identify.

- Many operations end up requiring top-level locks. The system loses some concurrency and task throughput is affected.

We tried to avoid those limitations when designing the data synchronization mechanism for the new prototype, and a lock server seems likely to have the same problem as the in-memory locks did. What is desired is a simplified lock acquisition procedure that makes deadlocks harder to happen and easier to identify, and improved task throughput.

This thesis explores a task scheduling approach instead of a lock server approach. Task scheduling requires all resources (or locks) be specified before task execution, which eliminates the possibility of having deadlocks in task execution code because tasks are not acquiring locks in the middle any more. It also has more control over when to start the tasks' execution, which could be leveraged to improve throughput. On the other hand, in order to use a task scheduler, tasks need to have well-defined resource requirements before they start execution. This can be cumbersome because tasks may need to resolve resource IDs before execution, possibly by querying the VIMS database. For example, a `VirtualMachine.powerOn` operation needs lock on the host that runs the virtual machine, and to obtain the host ID, one needs to query the database for the virtual-machine-to-host mapping. Table 1.1 summarizes the differences between the task scheduling approach and the locking approach.

| **Task Scheduling** | **Locking** |
|---|---|
| Eliminate deadlocks | Deadlocks are possible and may become tricky in a complex locking scheme |
| Tasks acquire all required resources as an atomic operation. Tasks do not block once dispatched from scheduler. | Tasks acquire locks one by one. They may block in the middle of execution holding locks to wait for other locks, which may hinder performance. |
| Tasks' resource requirements need to be fully specified before execution. | Tasks can specify locks during execution time. |

Table 1.1: Comparison of Task Scheduling and Locking Approach

## 1.2 Distributed VIMS Architecture

This section introduces the system internals of the distributed VIMS that the scheduler is designed for.

Distributed VIMS is a new prototype that was being developed at the time the thesis was made. The overall functionality is realized by a collaboration of different services (figure 1-1):

**API Service** Receives remote procedure calls from clients and executes them. Most remote procedure calls received by the API service are operations to be performed on one or more virtual infrastructure components, such as `VirtualMachine.powerOn` for powering on virtual machines, and `Host.destroy` for excluding the host from the cloud. Those operations are forwarded to the appropriate host machine(s). Changes to the components' states are written to database (managed by the inventory service. The database itself is not shown on graph).

**Inventory Service** Manages the database that stores information about all virtual machines, hosts and other virtual components in the cloud. All database reads and writes are made through the inventory service.

**Collector Service** Listens to updates from all host machines and updates the database accordingly.

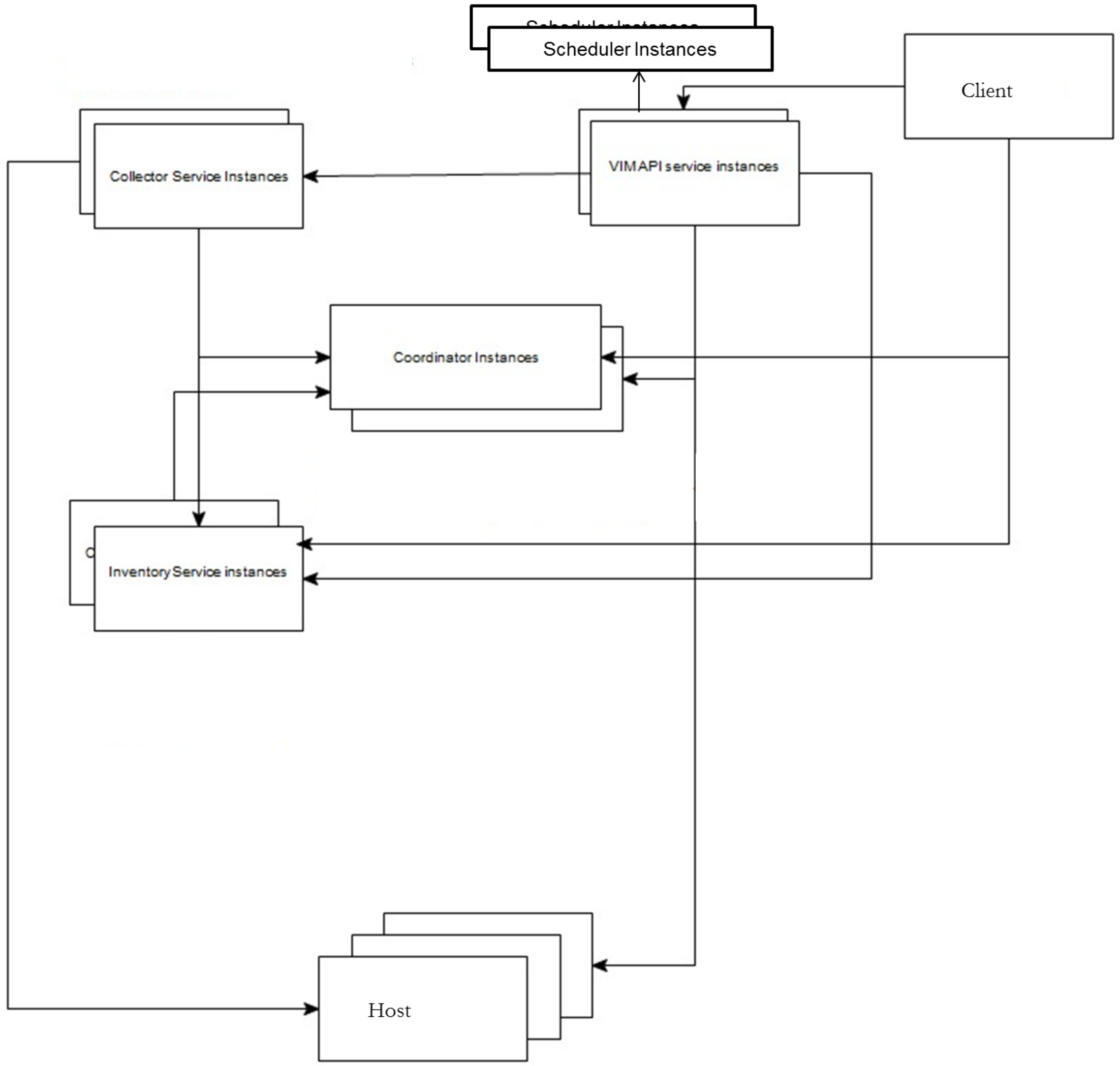


Figure 1-1: Software Architecture for Distributed VIMS

**Coordinator Service** Remembers IP addresses and other useful information for service discovery. Services register themselves with the coordinator at startup and retrieves information about other services through it.

Every service can have multiple instances running in parallel. The API service instances are identical. Client can choose any instance to send their remote procedure calls. Client knows about the services' existence through the coordinator.

### 1.2.1 Task Scheduler for Distributed VIMS

Task scheduler is a module that the API service communicates with to obtain required resources before executing each task (figure 1-1, above the API service instances). Once the task scheduler dispatches the task, it can be executed by the API service without worrying about getting locks to protect data access. The resources will be held by the task until its execution finishes.

## 1.3 Outline of Rest of Thesis

The rest of the thesis is organized as follows: Chapter 2 gives a brief survey on task scheduling algorithms and identifies the similarities and differences between the scheduling problems that these algorithms solve and the specific scheduling problem in VIMS. Chapter 3 describes the proposed scheduling algorithm used for task scheduling in VIMS. Chapter 4 describes the system design and implementation of the task scheduler in VIMS, distributed using Zookeeper as the data service and implementing a multi-instance failover scheme. Chapter 5 shows experimental results from tests that evaluate the scheduler's performance with respect to its scheduling efficiency, availability (as a function of crash rate) and scalability (the scheduler system overhead). Chapter 6 concludes the thesis by summarizing the contributions and listing future work recommendations.

# Chapter 2

# Related Work

Before introducing the task scheduler, it is helpful to define a few scheduling-related terms. A brief survey of basic offline and online scheduling algorithms is then provided. Finally, the characteristics of the scheduling problem for VIMS are defined in the same set of terminologies.

## 2.1 Terminology

Terminologies used in defining the scheduling problem are provided below.

**Offline Scheduling:** Also called static scheduling. Tasks' arrival times are know in advance. Scheduler computes a schedule offline before the tasks come.

**Online Scheduling:** Also called dynamic or real-time scheduling. Task arrival times are unknown and unpredictable. Scheduler only knows about the tasks when they arrive.

**Preemption:** Refers to the ability of suspending tasks and resuming them later possibly on a different machine [9].

**Precedence Constraints:** Specify tasks' predecessor relationships in a directed acyclic graph. Tasks cannot start before their predecessors finish [9].

**Conflicting Tasks:** Specify conflicting tasks in an undirected graph. Neighboring tasks cannot run at the same time [13]. (Unlike precedence constraints, the order of their executions does not matter.)

**Clairvoyant/Nonclairvoyant:** Clairvoyant scheduling means that the tasks' execution times are known to the scheduler, otherwise it is nonclairvoyant scheduling [12].

**Competitive Ratio:** A scheduling algorithm is $\sigma$-competitive if for each input instance the objective value of the schedule produced is at most $\sigma$ times larger than the optimal objective value [13, 12].

**Makespan:** The maximum task completion time among all tasks executed.

**Release Time:** Time when task is queued at scheduler.

**Completion Time:** Time when task finishes execution.

**Flow Time:** Time spent between release and completion, representing the response time of the server.

## 2.2 Scheduling Algorithms

A few basic algorithms are mentioned for offline and online scheduling. Online scheduling can be further divided into clairvoyant and nonclairvoyant scheduling.

### 2.2.1 Offline Scheduling

Offline scheduling algorithms mostly concern finding an optimal schedule given tasks' execution time, arrival time, precedence constraints, etc. Hu's algorithm [5] and Coffman-Graham Algorithm [3] produce optimal schedules for tasks with precedence constraints and equal execution times for minimizing makespan. The smallest-processing-time (SPT) algorithm schedules task with the smallest execution time first and is optimal for minimizing total completion time (sum of all task completion times) [9]. Hodgson-Moore algorithm [11] produces optimal schedules for tasks with deadlines and minimizes the number of late tasks (tasks that missed their deadlines) when a single machine is used.

### 2.2.2 Online Scheduling

Some basic clairvoyant algorithms are[1]:

- Shortest-remaining-processing-time (SRPT): Task with the shortest remaining execution time is run first, preempting current running tasks if necessary. The rule is optimal for average flow time on a single machine and provides the best-known approximation for parallel machines [1].

- First-in-first-out (FIFO): Task with the earliest release time runs first. It is optimal for minimizing maximum flow time on a single machine.

Some basic nonclairvoyant algorithms are[2]:

- Round-robin (RR): Each task runs for an equal amount of time in a circular fashion.

- Shortest-elapsed-time-first (SETF): Task that has been executed for the shortest amount of time is run first.

- Vertex coloring: This algorithm is for scheduling conflicting tasks with integral release times and either equal execution times or preemption. At any time it finds a coloring that uses the smallest number of colors on the available tasks' conflict graph (neighboring tasks cannot have the same color). It then dispatches tasks colored with one of the colors. It achieves a competitive ratio of 2 for minimizing makespan [13].

[1] The list is a subset of standard algorithms for *online-time* clairvoyant model in [12].

[2] The list is partially taken from standard algorithms for *online-time* nonclairvoyant model in [12].

### 2.2.3 Transaction Scheduling

Task scheduling has been studied in the context of database systems as transaction scheduling and concurrency control. Database transactions need to obtain shared locks on data reads and exclusive locks on data writes. Some of the concurrency control and locking protocols proposed are:

- Two phase locking (2PL): The most basic concurrency control protocol for synchronizing database transactions. A transactions only acquires locks in the expanding phase and only releases locks in the shrinking phase [2]. A transaction blocks if another transaction is holding the lock it needs.
- Tree protocol: Assuming data is organized into a rooted tree, a transaction can lock an entity in the tree only if its parent entity is locked by the transaction. A transaction can only lock an entity once. It can unlock an entity at any time. The protocol only allows exclusive locks [15].
- Protocols on directed acyclic graphs: A number of protocols have been proposed for data organized into a directed acyclic graph (DAG) [7, 14]. These methods restrict the order in which entities can be locked based on the structure of the DAG. They usually need to lock more entities than what is absolutely necessary (more than 2PL) [4].

## 2.3 Characteristics of Scheduling in VIMS

The characteristics of the scheduling problem for VIMS are listed below. Tasks in VIMS are remote procedure calls from clients with resource requirements, where each resource requirement protects the states of an infrastructure component such as a virtual machine, a host or a data center. Tasks' resource requirements are fully specified before execution.

- Online: Tasks are initiated from clients to the API service. Client requests are mostly aperiodic and aperiodic. The tasks' arrival times are unpredictable.

- Nonclairvoyant: The tasks' execution times are unknown. It is difficult to predict task execution times; they vary with operation types, operation targets and the load of the server machines.

- No Preemption: The API service does not support suspending and resuming ongoing tasks. But tasks can be aborted by the API service if they cannot be completed due to machine failures, network connection problems, etc. (Task information are stored in the VIMS database and will be properly updated in the case of an abortion by the API service. The scheduler will know that the task is aborted from a task tracker that reads tasks' status from the database.)

- No Hard Deadlines: Tasks do not have a deadline that they have to conform to. In fact, no deadlines (hard or soft) are specified for tasks in VIMS.

- Conflicting Tasks: Tasks conflict with each other if their resource requirements conflict.

- Non-uniform Machines: In VIMS, most tasks are first processed on the VIMS server, and then directed to the specific host machines that the task is targeted at. For example, a `VirtualMachine.powerOn` task is pre-processed by the API service, then forwarded to the hosting server of the virtual machine to perform the actual power on operation. Thus the task executions are typically spread out into host machines in the cloud. This is different from most of the online scheduling algorithms mentioned. They assume that tasks can run on any machine if there are multiples of them.

The scheduling problem in VIMS has some different characteristics from a pure online nonclairvoyant scheduling problem. The 2PL protocol mentioned in transaction scheduling is a possible way of allocating resources to tasks, but it does not take advantage of the predeclared resources and blocks just like in-memory locks. The tree protocol also does not take advantage of the predeclared resources and it is only for exclusive resources. The tree and the DAG protocols also assume a certain kind of resource organization representing resource granting restrictions (such as not granting a resource unless parent is granted), which does not necessarily correspond to the resource organization in VIMS. VIMS does not have such set of restrictions or rules on lock acquisition. The next chapter describes the algorithm that is used for scheduling in VIMS.

# Chapter 3

# The Scheduling Algorithm

This chapter describes the scheduling algorithm used by the scheduler for distributed VIMS.

## 3.1 Definitions

This section defines a few terms with respect to their specific meanings in VIMS.

**Task** A task refers to the operation requested by a remote procedure call from client, such as `VirtualMachine.powerOn`.

**Resource** A resource is a piece of data that can be read or written by more than one task. In VIMS, each virtual infrastructure component such as a virtual machine and a host is a resource. Resources can be divided into finer granularity and thus allow more concurrency.

**Ownership** Tasks specify resource requirements. A task can request *shared* (read-only) or *exclusive* (read/write) ownership of a resource. Multiple tasks can share a resource, while only one task can exclusively own the resource at a time.

**Task Scheduling** Given tasks with resource requirements, in real time dispatch tasks in an order that does not conflict in resource utilization.

## 3.2 Algorithm

The scheduling algorithm takes on a round-robin approach: Tasks are ordered by release time and their resource requirements are checked in a round-robin fashion. The ones whose resource requirements can be satisfied according to the current resource availability are scheduled. If there are multiple such tasks with conflicting requirements, the first one that the scheduler tries will be scheduled.

Figure 3-1 shows the pseudo code for the scheduling algorithm. The scheduler keeps a global record of the current resources' availability and checks each pending task's resource requirement against this record. If all the required resources for a task are available (i.e. the resource is not in use or is shared if the task requires *shared* ownership only), the resources are allocated to the task and the task is dispatched. If any of the required resources cannot be allocated to the task, the resources' availability is not changed (all previous resources allocated to the task will be reverted).

### 3.2.1 An Example

We provide an example of scheduling four tasks using the algorithm. The resource requirements of the four tasks are shown in table 3.1. Names such as cluster-1, host-1 and vm-2 are resource IDs. Resource requirements are specified with resource ID followed by the requested ownership. For example, dc-1:S means the task requires *shared* ownership of resource dc-1. In a real setting, the resource requirements are manually specified for each API method served by the API service (as part of the specification of the API) using a descriptive language, which is then parsed by the API service. Resource IDs are obtained possibly by querying the database through the inventory service. For example, to obtain the IDs of the host, the cluster and the data center (dc) that contain the virtual machine, the API service needs to query the database for virtual-machine-to-host, host-to-cluster, and cluster-to-data-center mappings.

Resource requirements of the four tasks can be shown in a conflicting tasks graph, where neighboring tasks cannot be run at the same time. (For definition of conflicting

```
List<Task> tasks; // array of pending tasks

// resource availability, maps resource id to resource
Map<String, Resource> resources;

void schedule() { // main scheduling loop
   for (Task task: tasks) {
      if (trySchedule(task)) { // succeed
         dispatch(task);
         tasks.remove(task);
      }
   }
}

boolean trySchedule(Task task) {
   // Resources  are sorted to avoid deadlock
   List<Resource> required = task.getSortedResources();

   List<Resource> occupied = new List<Resource>();

   // Acquire resources *in order* to avoid deadlock
   for (Resource reqr : required) {
      Resource r = resources.get(reqr.id);

      if (r.ownership().isEmpty() ||
          reqr.ownership().isShared() && r.ownership().isShared()) {
         // Modify resource availability to indicate task owns resource
         // with proper ownership right.
         r.occupy(task, resource.ownership());
         occupied.add(r);
      } else {
         // Failed, revert all previous occupied resources
         for (Resource r : occupied) {
            r.free(task);
         }
         return false;
      }
   }

   return true; // succeed
}
```

Figure 3-1: Scheduling Algorithm Pseudo Code

| Task ID | Operation | Resource Requirement |
|---|---|---|
| T1 | `vm-1.powerOn` | dc-1:S, cluster-1:S, host-1:S, vm-1:E |
| T2 | `vm-2.powerOn` | dc-1:S, cluster-1:S, host-2:S, vm-2:E |
| T3 | `cluster-1.destroy` | dc-1:S, cluster-1:E |
| T4 | `host-1.destroy` | dc-1:S, cluster-1:S, host-1:E |

Table 3.1: Resource Requirement of Example Tasks

tasks, see section 2.1, 2.3.) Figure 3-2 shows that task T3 conflicts with all the other tasks because the resource cluster-1:E conflicts with all the other tasks' resource cluster-1:S, and T1 conflicts with T4 because they require host-1:S and host-1:E respectively. If we perform a vertex coloring on the graph such that no two nodes sharing the same edge have the same color, the graph can have three colors with T1 and T2 sharing the same color.

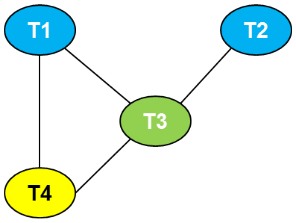


Figure 3-2: Conflicting Tasks Graph

Figure 3-3 shows the timeline for changes in resource availability and the four tasks' lifetime from release to completion.

All four tasks are queued at startup time t0. Suppose they are added to the pending task queue in the order of T1–T4. Scheduler checks resources availability for each task in the queue from the first one to the last, and then starts from the first one again.

Task T1 and T2 obtain ownerships to their required resources first and start execution at time t1. This prevents task T3 and T4 from obtaining resources at the same time. At time t2, task T1 finishes execution and the occupied resources are freed. This allows T4's resource requirements to be satisfied and task T4 starts running. At time t3, task T2 finishes execution while T4 is still running. At time t4, task T3's resource requirements is satisfied after T4 frees up resource *cluster-1*.

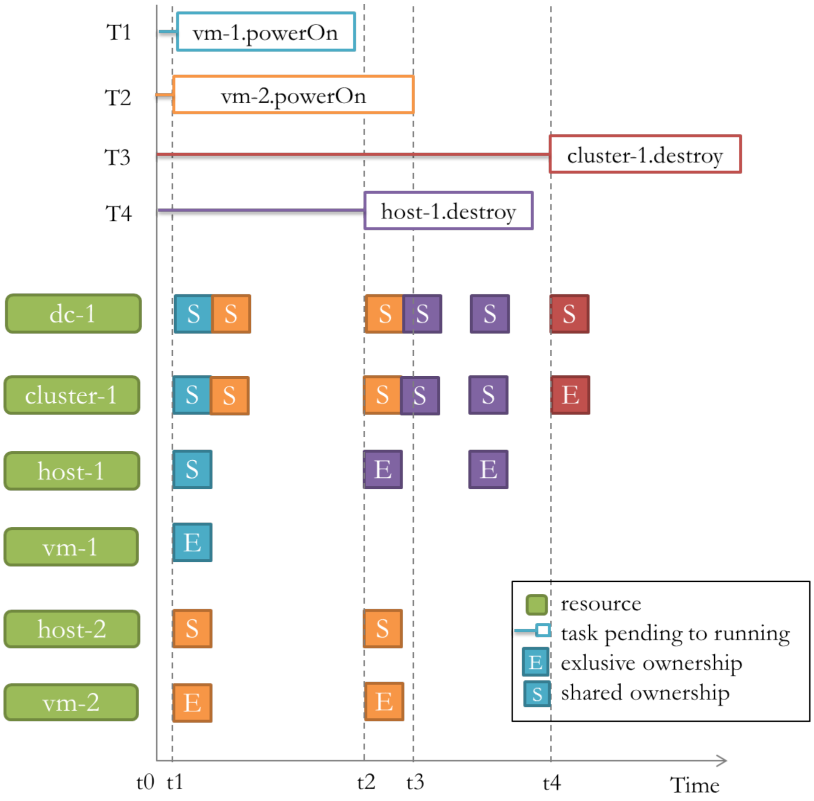


Figure 3-3: Task Scheduling Example

# Chapter 4

# System Design and Implementation

As a component in distributed VIMS, like other services, the scheduler must have high availability, i.e. one instance failure should not stop the scheduling service and failed scheduler instances should be able to restore their states.

This chapter describes the design and implementation of the distributed scheduler module for VIMS. The distributed scheduler's Zookeeper-based data services, Zookeeper-based scheduling algorithm, task lifetime within the scheduler, and the failover mechanism for high availability are explained in detail.

## 4.1 Architecture

The scheduler service is a multiple-instance service with a shared data service (Zookeeper) where tasks and resource availability information are kept. The multiples instances are identical, API service can send tasks to any instance.

Figure 4-1 shows the scheduler module and services that it interacts with in the VIMS system.

`Task` and `Resource` are data objects that each contains an unique ID. `Task` includes in addition a list of `Resource` representing its resource requirement. It can also include additional information that may not be used by the scheduler, such as client connection

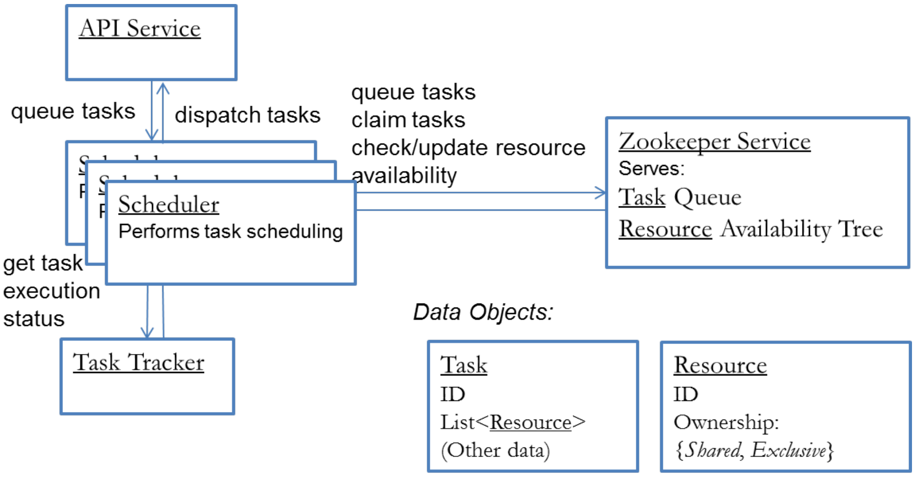


Figure 4-1: Scheduler System Architecture

information used by API service for sending task results back to clients. `Resource` contains an `Ownership` that can be either *shared* or *exclusive*, specifying whether the resource is currently shared or exclusively owned when it is occupied by some task(s).

Scheduler receives incoming tasks from the API service and sends back dispatch calls to it when tasks are ready to run. The steps taken after API service receives a remote API call are as follows:

1. API service constructs a `Task` object, including the required resources' IDs and requested ownerships;

2. API service informs scheduler about the new task;

3. Scheduler adds the task to the task queue;

4. When the scheduler has successfully acquired resources for the task, it informs API service to dispatch the task. Task tracker starts tracking the task's execution status;

5. API service executes the task that reads and/or modifies system states;

6. Scheduler is informed when the task finishes execution by task tracker. It then releases resources occupied by the task.

Task tracker tracks tasks' execution status when they start. Scheduler relies on task tracker to figure out when the task's occupied resources can be freed.

Scheduler uses Zookeeper as the shared data service that stores the task queue and the resource availability information. All scheduler instances work on the same shared task queue and resource availability data through Zookeeper.

## 4.2 Zookeeper-based Data Structures

Zookeeper is an open-source coordination service that is suitable for maintaining small pieces of data such as configuration and naming information for distributed applications. The service can be served from multiple Zookeeper servers that form a quorum. The servers agree with each other by running an algorithm similar to Paxos that dynamically selects a leader. The service will keep running as long as the majority of servers in the quorum are alive [6]. Clients to Zookeeper (in our case, the scheduler instances) can connect to any server for reads and updates .

In Zookeeper, data is organized into a tree of nodes. Each node has a name, an absolute path from the root and stores data with it. Figure 4-2 shows the organization of the node tree used as the task queue and resource availability records.

There are three type specifiers for a node that Zookeeper defines which can be set at creation time:

**Ephemeral** Ephemeral nodes exist during the active life time of the client session that created it. They will be removed if the client disconnects from Zookeeper.

**Permanent** Opposite to ephemeral nodes, permanent nodes will not be automatically deleted. They will exist until some client requests a deletion.

**Sequential** When created, sequential nodes' names will be appended with an integer, which is a monotonically increasing counter that counts the number of children under the created node's parent. Each sequential node created under the same parent will be assigned a unique value that indicates their order of creation.

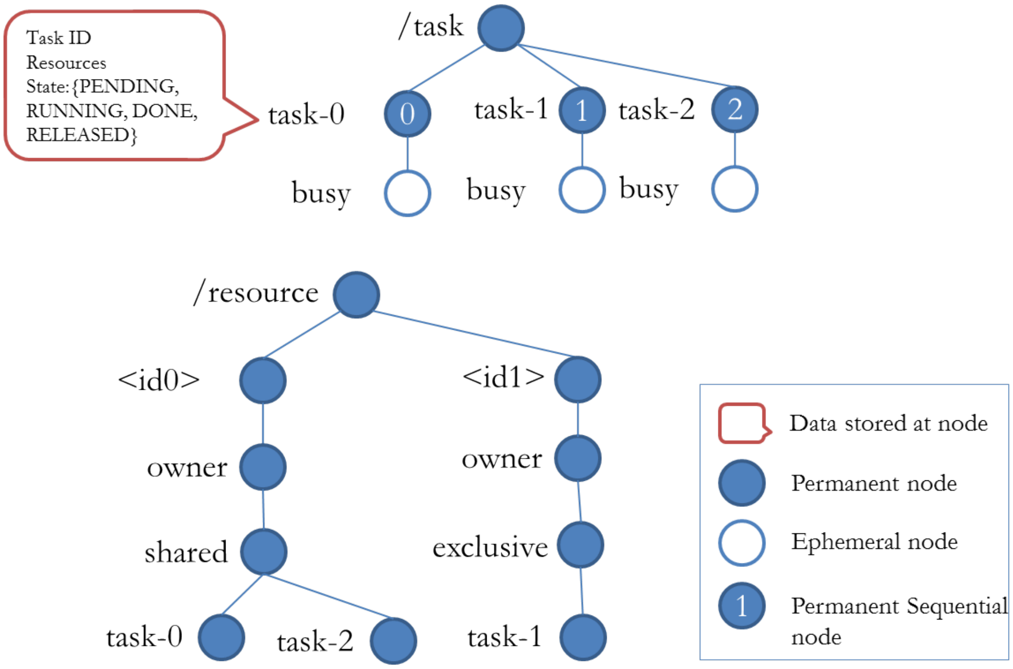


Figure 4-2: Data Organization of the Task Queue and Resource Availability Records in Zookeeper

The next two subsections describe the data organization of the task queue and the resource availability records and how they are used by the scheduler in detail.

### 4.2.1 Task Queue

The task queue consists of a root node with path `/task` and a number of child nodes under it, each representing a task (figure 4-2 top). Task nodes are created as sequential nodes by the scheduler instance that receives the task from the API service. The `Task` object is serialized into a string and stored at the task node. The integer appended to the nodes' names is used to determine the order of the tasks. Tasks that are created earlier (i.e. have a smaller integer value) will be checked earlier against resource availabilities.

Scheduler instances take tasks from the queue and create a `busy` node under the task node they are taking. It always takes the task with the smallest integer value that does not already have a `busy` node. When a task is taken by a scheduler instance, it adds the task to its local queue, which it loops through to check resource requirements against the resource tree (described in the next subsection 4.2.2).

The purpose of creating a busy node instead of deleting the task node from the shared queue is to remember the tasks in case of instance failure. When the scheduler instance that holds the task in its local queue crashes or loses network connection, the busy node will be deleted but not the task node. This enables another instance to take over the tasks that would have been lost if they were removed from the shared queue when they are taken by an instance. (For a complete description of the scheduler's failover scheme, see section 4.4.)

### 4.2.2 Resource Tree

The resource tree contains the current resource availability information (figure 4-2 bottom). It has a root node with path /resource. Each resource is a node the resource tree root, with the resource ID as node name. If the resource is owned by one or more tasks, node owner will be created under the resource node, with a child node named either shared or exclusive depending on the ownership type that the task(s) require. The type node has one or more children that each corresponds to a task, with the same name as the corresponding task node in the task queue, indicating the owner the resource. Node owner/shared can have more than one child, while node owner/exclusive can only have one.

Resource is released by deleting all nodes under the resource node. A task releases its *shared* ownership, in the case of multiple sharers, by only removing the leaf node that corresponds to itself .

## 4.3 Zookeeper-based Scheduling Algorithm

The Zookeeper-based scheduling algorithm implements the round-robin scheduling approach as described in chaper 3, but the loops are around local queues at each scheduler instance, not on the Zookeeper-based global task queue.

The algorithms deal with Zookeeper-based data synchronization properly and can be run by multiple scheduler instances in parallel, without corrupting the task queue or resource tree states. (For example, the case when node `owner/shared` and node `owner/exclusive` both exists, or multiple nodes exist under `owner/exclusive` will not happen.) No explicit locks on the resource tree or task queue are used in this algorithm; synchronization is done by making careful use of Zookeeper's inherent consistency guarantees and properties.

Pseudo code of three procedures: taking task from queue, acquiring resources for a task and releasing resources taken by a task are shown and explained. Zookeeper's consistency guarantees are described next, followed by arguments for the correctness of the algorithms.

Figure 4-3 shows the pseudo code for taking tasks from the task queue. The procedure sorts the task nodes by the integer assigned at creation time. It then tries to take the first task in the sorted list (the one with the smallest integer value) by creating a `busy` node under the task node (line 12). If the creation is successful, it has successfully taken the task. If not, it goes on to try the next smallest one.

```
 1 Task grabTask() {
 2     // sort the task names by the integer assigned by Zookeeper
 3     List<String> orderedTaskNames = sortByIntegerSuffix(
 4         Zookeeper.getChildren(”/task”));
 5     if (orderedTaskNames.isEmpty()) {
 6         return null;
 7     }
 8     for (String node: orderedTaskNames) {
 9         String taskPath = ”/task/” + node;
10         try {
11             // create busy node
12             Zookeeper.create(taskPath + ”/busy”);
13             // construct Task object from stored data
14             Task t = Task.deserialize(Zookeeper.getData(taskPath));
15             return t; // succeed
16         } catch (NodeExistsException e) {
17         } catch (NoNodeException e) {
18         }
19     }
20     return null;
21 }
```

Figure 4-3: Pseudo Code for Taking Tasks from Zookeeper-based Task Queue

Figure 4-4 shows the pseudo code for declaring ownerships of the required resources for a task. For each resource that the task requires, the procedure first makes sure that the resource ID node exists, then checks whether node `owner` exists. If it does not exist, the scheduler will try to create it and all the necessary nodes under it (as described in section 4.2.2): node `owner`, node `owner/exclusive` or `owner/shared`, and a leaf node with the same name as the task node (line 23–25). If node `owner` already exists, the scheduling for the task is unsuccessful, except when an `owner/shared` node exists *and* the current task requires *shared* ownership. In this case, the scheduler just needs to create a child node under the node `owner/shared` (line 29–32).

The node creation operations for all required resources of a task are performed as an atomic operation. This is realized by the Zookeeper's multi operation (line 41). The multi operation guarantees that either all the operations specified are applied successfully or none of them is applied [16]. This saves us the effort of having to revert the changes made to previous successfully occupied resources if the resource allocation for the task fails in the middle.

Figure 4-5 shows the pseudo code for releasing resources occupied by a task when it has finished execution. Resources are released by deleting all the nodes under the resource node, except for shared resources with additional sharers, only the leaf node that belongs to the task is deleted.

### 4.3.1 Algorithm Correctness

The correctness of the algorithms relies on a number of Zookeeper consistency guarantees. Zookeeper has the following basic consistency guarantees [18]:

- All Zookeeper operations such as node creation, deletion and the multi operation (multiple operations grouped in a single transaction, see figure 4-4) are atomic. I.e. updates either succeed or fail; there are no partial results.
- Updates from a client will be applied in the order that they were sent.

Zookeeper has the following specifications regarding node creation [16]:

- Nodes under the same parent cannot have the same name. Attempt to create a node that has the same name as an existing node under the same parent will

```
boolean trySchedule(Task task) {
  // Zookeeper operations in this list either all execute
  // or none of them executes
  List<Op> ops = {};

  for (resource: task.resources) {
    // path to resource id node
    resourcePath = ”/resource/” + resource.id;
    // path to owner node
    ownerPath = resourcePath + ”/owner”;
    // path to ownership (shared or exclusive) node
    ownershipPath = ownerPath + (rsc.isShared()?”shared”:”exclusive”);
    // path to leaf node (the task)
    taskPath = ownershipPath + task.nodeName;

    // make sure resource id node exists
    if (!resourcePath.exist()) {
      createNode(resourcePath);
    }

    if (!ownerPath.exist()) {
      // no owner node, create all necessary nodes under
      ops.add(Op.create(ownerPath));
      ops.add(Op.create(ownershipPath));
      ops.add(Op.create(taskPath));
    } else if (taskPath exist) {
      // already owns this resource
      continue;
    } else if (resource.ownership == shared
          && ownershipPath.exist()) {
      // requires shared ownership on a shared resource
      ops.add(Op.create(taskPath));
    } else {
      // cannot own the resource
      return false;
    }
  }

  // Executes all node creations or none of them
  try {
    Zookeeper.multi(ops);
    // all node creations succeeded
    return true;
  } catch (KeeperException e) {
    // one or more node creations fail,
    // none of the node creations takes effect
    // schedule fails
    return false;
  }
}
```

Figure 4-4: Pseudo Code for Task Resource Allocation on Zookeeper-based Resource Tree

```
void release(Task task) {
   for (Resource resource: task.resources) {
      // path to owner node
      ownerPath = ”/resource/” + resource.id + ”/owner”;
      // path to ownership (shared or exclusive) node
      ownershipPath = ownerPath + (rsc.isShared()?”shared”:”exclusive”);
      // path to leaf node (the task)
      taskPath = ownershipPath + task.nodeName;

      if (Zookeeper.exist(taskPath)) {
         Zookeeper.delete(taskPath);
      } else {
         // resource has been released for this task
         continue;
      }

      if (resource.ownership == exclusive
         || Zookeeper.getChildren(ownershipPath).isEmpty()) {
         try {
            Zookeeper.delete(ownershipPath);
         } catch (NotEmptyException e) {
            // another task added itself as a sharer first
            continue;
         } catch (NoNodeException e) {
            // another task added itself as a sharer first
         }
      }

      try {
         Zookeeper.delete(ownerPath);
      } catch (NoNodeException e) {
         // another finished sharer deleted it first
      }
   }
}
```

Figure 4-5: Pseudo Code for Releasing Resources on Zookeeper-based Resource Tree

result in a NodeExists error.

- Nodes cannot be created if its parent node does not exist. Attempt to create a node whose parent node does not exist will result in a NoNode error.

Based on the above Zookeeper guarantees, the following arguments can be made for the correctness of the Zookeeper-based scheduling algorithm:

- **No tasks can be taken by more than one scheduler.**

  A task is taken by the scheduler successfully creating the `busy` node under it in the task queue. If several schedulers are trying to take the same task, only one will succeed in creating the `busy` node and the others will fail and receive a NodeExists error. The `busy` node is ephemeral (see section 4.2 for a description of ephemeral nodes), and will be automatically deleted by Zookeeper if the scheduler instance that created it loses connection to Zookeeper servers. Scheduler instances that fail will not be able to continue making changes on Zookeeper because their client sessions established with the Zookeeper servers end when the connection is lost (for failure detection see section 4.4.2).

- **No conflicting resource ownerships can be created in the resource tree.**

  Two resource ownerships conflict if they are for the same resource and at least one of them is an *exclusive* ownership.

  A task owns a resource by the scheduler successfully creating node `owner`, or alternatively, appending a node with the task name to node `owner/shared` if the task requires *shared* ownership.

  This can be seen by enumerating the possible outcomes of tasks resource acquisition while the resource is in the following four possible states: (For the resource tree nodes organization, see figure 4-2.)

  - *For resources in shared ownership (i.e. having node `owner/shared`), tasks that require exclusive ownership will fail to own the resource. Tasks that require shared ownership can succeed.*

For tasks that require *exclusive* ownership, the attempt to create the `owner` node will result in a NodeExists error. For tasks that require *shared* ownership, it can successfully create a child node under `owner/shared`[1].

- *For resources in exclusive ownership (i.e. having node `owner/exclusive`), all tasks that require the resource will fail.*

  For tasks that require *exclusive* ownership, the attempt to create the `owner` node will result in a NodeExists error. For tasks that require *shared* ownership, the attempt to create the `owner` node will result in a NodeExists error; the attempt to create a child node under `owner/shared` will result in a NoNode error.

- *For resources that are not in use (i.e. not having the `owner` node), among all tasks that require the resource, only one will succeed.*

  Tasks that require *exclusive* ownership will try to create the `owner` node. Tasks that require *shared* ownership will either try to create the `owner` node or create a child node under `owner/shared`. Since the `owner` node does not exist, the attempt to create a child node under `owner/shared` will result in a NoNode error. Only one of the attempts to create the `owner` node will succeed.

- *For resources that are in a transient state (i.e. having the `owner` node with no child nodes. This can occur in the middle of releasing resources), all tasks that require the resource will fail.*

  Attempts to create the `owner` node will result in a NodeExists error. Attempts to create a child node under `owner/shared` will result in a NoNode error.

Thus the resource tree remains correct in all four possible states with all possible scheduler ownership declaration actions. The resource tree thus cor-

[1] Tasks that try to acquire *shared* ownership by creating the `owner` node will not succeed and will get a NodeExists error. It is possible because the scheduler instance may not have seen the `owner/shared` node when it last checked the existence of the `owner` node. I.e. the resource tree could be changed between when the scheduling procedure last checks (line 21, figure 4-4 ) and when the node creation operation happens (line 41, figure 4-4)

rectly reflect the current resource utilizations, no invalid states (such as node `owner/shared` and node `owner/exclusive` both exists, or multiple nodes exist under `owner/exclusive`) cannot happen.

## 4.4 Availability

This section describes the failure detection and failover scheme in a multiple-instance scheduler. The scheduler service will keep running as long as Zookeeper has a majority of servers in the quorum running, and at least one scheduler instance is running.

### 4.4.1 Task Lifetime

Scheduler keeps task state information at the task node in task queue (figure 4-1) for use in failover. This section describes the state changes of tasks during its lifetime as seen by the scheduler.

To the scheduler, the lifetime of a task starts when the API service sends the task to a scheduler instance. It ends when all resources the task owns are released in the resource tree. Task goes through four states during its lifetime (figure 4-6):

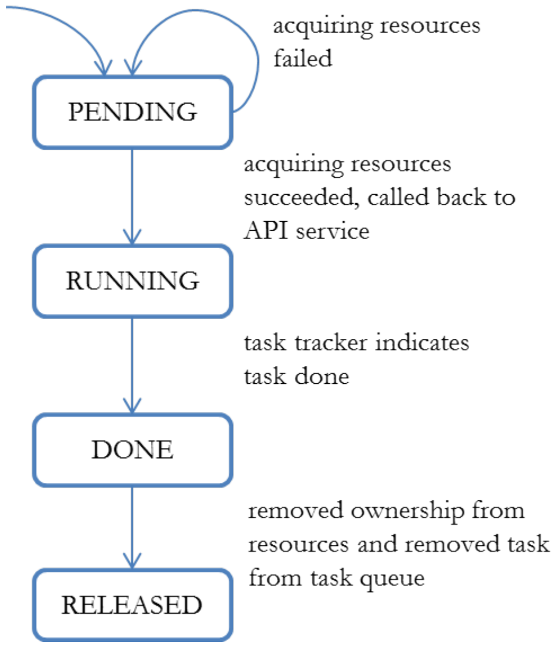

acquiring resources failed
PENDING
acquiring resources succeeded, called back to API service
RUNNING
task tracker indicates task done
DONE
removed ownership from resources and removed task from task queue
RELEASED

Figure 4-6: Task States

**PENDING** All tasks start in the PENDING state. This state corresponds to the period of time when the scheduler is trying to acquire resources for the task but has not yet succeeded. Task enters PENDING state when it gets in the task queue, and remains in this state until all its required resources ownerships are acquired and the scheduler has informed the API service to dispatch the task.

**RUNNING** Task enters the RUNNING state after the scheduler has informed the API service to dispatch the task. Task remains in the RUNNING state until the task tracker indicates that the task has finished execution. The task execution can succeed, fail or be canceled. In all three cases the task's execution is considered done.

**DONE** Task enters the DONE state after the task tracker indicates that it has finished execution. It remains in this state until all the resource ownerships it acquired have been released in the resource tree and its corresponding task node has been removed from the task queue.

**RELEASED** Task enters the RELEASED state after the task node has been removed from the task queue. The task is fully completed at this stage and the relevant information about the task stored at the task node are deleted when the task node is deleted.

When state changes, the scheduler instance that took the task will update the state information stored at the task node.

### 4.4.2 Failure Detection

Scheduler relies on the property of Zookeeper's ephemeral node for detecting instance failures. An ephemeral node will be automatically deleted if the scheduler instance that created it loses connection to the Zookeeper servers. A `busy` node is created as an ephemeral node by the scheduler instance that takes the task. Once a scheduler instance loses connection with Zookeeper, all `busy` nodes it has created so far will be removed and its client session established with Zookeeper ends. Any operation that the scheduler instance performs on Zookeeper after the client session ends will fail and result in a client-session-expired error. If the scheduler instance is restarted later, it has to establish a new client session with Zookeeper.

When the `busy` nodes created by the instance disappear after instance failure, the tasks that were once taken by the failed instance will then be seen as not taken and will be taken by other scheduler instances. The next section describes how the left-over tasks that can be at different states in their lifetime are processed.

### 4.4.3 Failover Scheme

The differences between left-over tasks (tasks that was once taken) and new tasks in the queue are their states. For a new task that is just added to the queue, the state is always PENDING, while for a task whose was taken by a failed scheduler instance, the task may be left in any of the four possible states.

A simple failover scheme that relies on the tasks' state information stored in Zookeeper is implemented. After scheduler takes a task from the task queue, it decides what to do based on the task's state. Table 4.1 shows a brief summary of the scheduler's failover actions. The details of the failover situation for each state are discussed below.

| **State** | **Action** | **Caveats** |
|---|---|---|
| PENDING | Normal scheduling procedures for new tasks | API service needs to deal with duplicate dispatch calls |
| RUNNING | Check task execution status with task tracker and release resources when task done | Task execution status has to be available regardless of scheduler failures. |
| DONE | Release resources for the task | None |
| RELEASED | Remove task from task queue | None |

Table 4.1: Failover Scheduler Actions

#### Task in PENDING state

There are three possible cases that will result in task left in the PENDING state:

1. The resource tree has not been updated for the task, i.e. the task has not acquired its resources yet.[2] In this case, the task has no difference from a new task that has not yet been taken by any scheduler.

2. The resource tree has been updated, but the dispatch call to the API service has not been sent.

3. The dispatch call has been sent to the API service, but the state at the task node has not been updated yet.

When scheduler gets a task in the PENDING state, it runs the normal procedures as if it is a new task, starting from resource acquisition (figure 3-1). The procedure checks whether resources have been claimed by the task and will succeed if the resource tree is already properly updated for the task. The dispatch call is then made.

Note that dispatch will be called twice if the left-over task is in case 3. This could be a problem if there are no checks for duplicate tasks before execution and the task is not idempotent. The problem should be properly dealt with by the API service that receives the dispatch calls. In VIMS, tasks are stored in the database when they are created by the API service, and the tasks' execution status are updated either by the API service or by the collector service that listens to updates from hosts (which include updates to task executions) during its execution process. The API service checks tasks' existence before they dispatch the tasks. If the task already exists in the database, it will not be dispatched again.

### Task in RUNNING state

Tasks in RUNNING state are already dispatched. The scheduler just needs to ask task tracker for the task execution status. Task tracker gets this information either from API service or directly from the database.

[2] The resource tree is either updated to reflect all the task's resource ownership requirements or is not updated at all for the task. This is guaranteed by the Zookeeper *multi* operation performed by the scheduling procedure (section 4.3).

### Task in DONE state

Tasks in DONE state have finished execution and may have already released some or all of its occupied resources. Scheduler in this case will execute the resource releasing procedure (figure 4-5). The procedure will ignore resources that are already released.

### Task in RELEASED state

Tasks in RELEASE state has released all the resources it occupies and are ready to be removed from the task queue.

# Chapter 5

# Experimental Results

This section evaluates three hypotheses about the scheduler:

1. Scheduling efficiency: Does the scheduler perform better than the lock-based approach? Does it perform worse than the optimal schedule if the optimal schedule is known?

2. Availability: Does the running time increase only linearly with crash rate?

3. Scalability: Does the scheduler add only a small overhead to the system?

Four sets of tests were run to evaluate the hypotheses. The first two benchmark tests compare the overall task completion time of the scheduler with the optimal schedule (if known) and the completion time of the lock-based approach. The third test measures how performance degrades with respect to crash rate. And the fourth test measures how performance is affected by running multiple scheduler instances or Zookeeper servers.

## 5.1 Environment

All tests were run on a single computer. Table 5.1 lists the specifications of the machine used.

| **Component** | **Specification** |
|---|---|
| Processor | 2.10Ghz Intel Core2 Duo processor (T6500) |
| RAM | 4GB |
| Disk | 5400 RPM SATA |
| OS | Windows 7 64-bit |

Table 5.1: Machine Specifications

The VIMS prototype and the scheduler are both developed in Java (version 1.6) and use Zookeeper Java library version 3.4.3. All tests are run with 512MB JVM stack size.

### 5.1.1 Simulating Lock Server

A server that simulates the lock granting behavior was implemented using Zookeeper for an approximate comparison of the relative efficiency of the lock server and the scheduler. Instead of obtaining ownerships on all required resources of a task at once (which is what the scheduler does), the simulated lock server obtains ownerships for the task's required resources one by one. It does not revert the ownerships obtained when a later resource acquisition fails. Tasks' required resources are sorted by resource ID before resource acquisition to avoid deadlocks.

Using the simulated lock server, tasks acquire resources one by one, waiting for new locks while holding old locks, which is similar to using a real lock server. The differences between them is that using the simulated lock server, tasks still have to acquire all locks before execution can start, while using a real lock server, tasks are not required to acquire all locks at the beginning.

## 5.2 Exclusive-Only Rectangular Benchmark

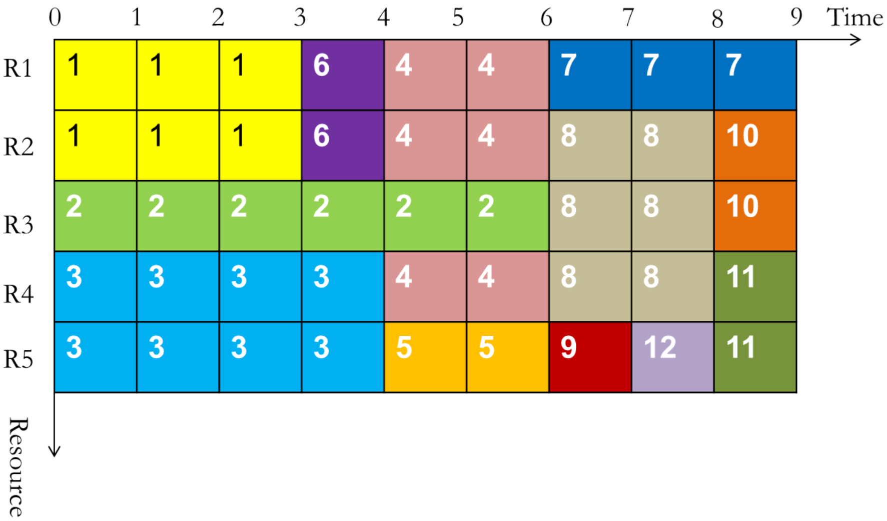
This benchmark generates tasks that acquire only *exclusive* ownership and provides a known optimal schedule that minimizes completion time. (Completion time is defined as the time it takes for all tasks to finish execution.) The benchmark is used to evaluate the scheduler's scheduling efficiency compared to the optimal schedule and the simulated lock server.

### 5.2.1 Benchmark

#### Task Generation

Tasks are generated with a known optimal schedule that has minimal completion time. If we construct a 2D space to represent tasks with the horizontal axis representing time and the vertical axis representing resources, the benchmark is generated by packing tasks into a rectangle. The benchmark generates tasks that acquire only *exclusive* ownership so that the tasks do not overlap, which simplifies the rectangle filling process. Figure 5-1 shows an example of the generated tasks. In the graph every colored rectangle represents a task and is labeled with the task's ID number. The left edge of a task rectangle marks the start time of the task in the optimal schedule, and the task rectangle spans the length of its execution time. The width (along the vertical axis) of the rectangle indicates the resources the task needs. For example, the yellow area labeled with number 1 represents task 1 which starts at time 0, occupies resources R1 and R2 and ends at time 3. The task's queue time is not shown and is a randomly generated number smaller than the start time. (For tasks with start time 0, queue time is 0.) There are 5 resources (R1–R5) in the graph. 11 tasks are generated. The optimal schedule takes 9 time units to execute. The generated schedule is optimal because all resources are being used at every moment in time.

To generate the example benchmark shown in figure 5-1, the generating procedure first creates task 1, 2 and 3 that collectively occupies all the resources at the starting edge (time = 0). Then it takes the maximum length of the three tasks as the length of the rectangle to be formed. In this case the length of the to-be-formed rectangle is 6. Task 4, 5 and 6 are then generated to fill the rectangle. Now there is a uniform starting edge at time 6 and the process can repeat to create a new rectangle. In the graph the second iteration generates task 7–12 that form a rectangle of length 3. All rectangles span all the resources, i.e. they all have the same width (5 in the example).



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| R1 | 1 | 1 | 1 | 6 | 4 | 4 | 7 | 7 | 7 |
| R2 | 1 | 1 | 1 | 6 | 4 | 4 | 8 | 8 | 10 |
| R3 | 2 | 2 | 2 | 2 | 2 | 2 | 8 | 8 | 10 |
| R4 | 3 | 3 | 3 | 3 | 4 | 4 | 8 | 8 | 11 |
| R5 | 3 | 3 | 3 | 3 | 5 | 5 | 9 | 12 | 11 |

Figure 5-1: Example of Tasks Generated in Rectangular Benchmark
Benchmark parameters: maximum task execution time = 6, number of resources = 5, number of rectangles = 2.

### Benchmark Parameters

The following parameters can be set to generate task sets that have different number of tasks and resources:

**Maximum Task Execution Time:** The longest running time of a task (which does not include queue time when task is waiting for resources). The benchmark generator assigns each task a random value between 1 and this number to be its running time (except for those tasks that are constrained by the rectangle boundaries).

**Number of Resources:** Defines the width of the rectangle. The benchmark generator assigns each task a random number between 1 and this number (possibly decreased to fulfill the rectangle requirement) to be the task's required number of resources.

**Number of Rectangles:** The number of rectangles to be generated. A larger number of rectangles imply more tasks and longer total execution time.

### 5.2.2 Procedure

The benchmark was tested on both the scheduler and the simulated lock server. A task simulator releases each task's at their precomputed queue time and keeps track of tasks' execution times after tasks are dispatched.

Table 5.2 lists values of the parameters set in the tests, the independent variables chosen and the dependent variables that are measured.

| **Benchmark Parameters** | |
|---|---|
| Maximum task execution time (seconds) | 20 |
| Number of resources | 100 |
| Number of rectangles | 1–10 |
| **Variables** | |
| Design | Optimal, Scheduler, Lock |
| Number of tasks | Varies for each generation and depends on the benchmark parameters set |
| **Measures** | |
| Completion time (seconds) | Time spent for all tasks to finish execution |

Table 5.2: Experimental Setup for Rectangular Benchmark Tests

#### Test 1: Comparing Three Designs

The benchmark test sets are generated with number of rectangles set to 1, 2 and 3, each generating 3 task sets. Each task set runs on both the scheduler and the simulate lock server.

#### Test 2: Comparing Scheduler to Optimal

The benchmark test sets are generated with number of rectangles set to 1, 3, 5, 7 and 10, each case generating 3 task sets. Each task set runs only on the scheduler.

### 5.2.3 Results

#### Test 1: Comparing Three Designs

Analysis of variance showed a significant main effect for type of design (optimal, scheduler or lock) on completion time ($F_{2,17} = 13.38$, $p = 0.0003$). Multiple comparison analysis confirmed that the lock-based approach has longer completion time than the other two. (For details, see appendix A, section A.1.)

Figure 5-2 plots the completion time of all three designs as a function of the number of tasks. The nine data points can be viewed as in three groups, each corresponds to a different number of rectangles generated.

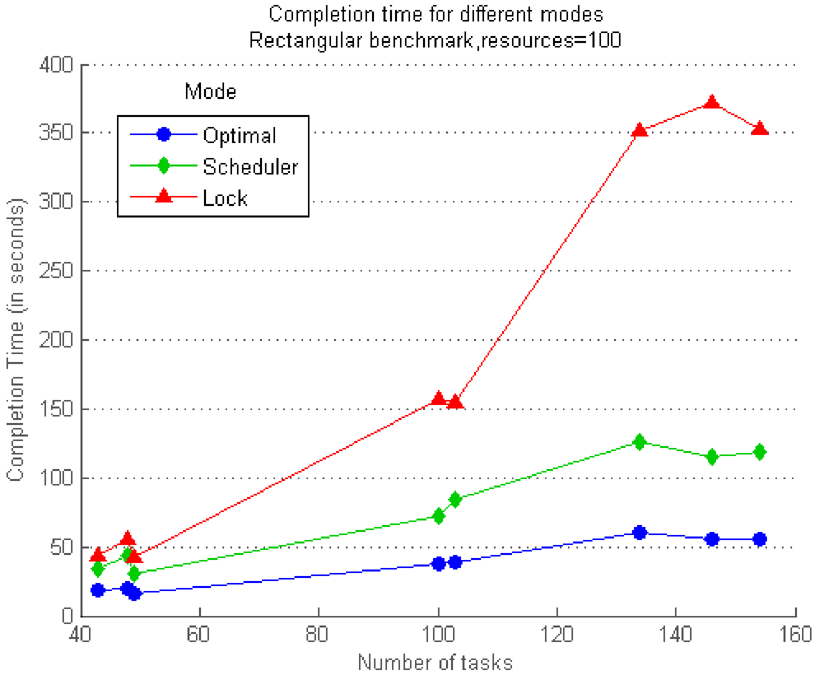


Figure 5-2: Completion Time for Different Designs by Number of Tasks (Rectangular Benchmark). All data points in the data set are shown.

Figure 5-3 shows the ratio of scheduler's and lock-based approach's completion time to optimal.[1] Completion time of scheduler is on average 2.10 times longer than optimal ($mean = 2.0975$, $stderr = 0.2528$). Completion time of the lock-based approach is on

[1] Linear fitting of the scheduler to optimal ratio is not a good fit ($r^2$ is small). The linear fittings are simply to provide a convenient visual aid for understanding how the ratios change with the number of tasks.

average 4.32 times longer than optimal ($mean = 4.3197$, $stderr = 0.5492$). The plot shows roughly that the lock to optimal ratio increases more rapidly than the scheduler to optimal ratio. (It is unclear how the scheduler to optimal ratio grows, more...)

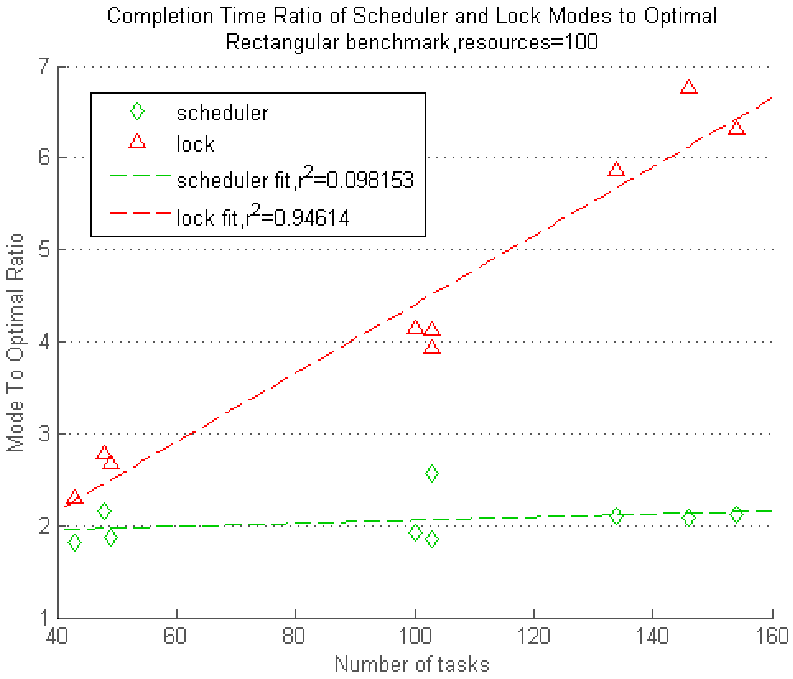


Figure 5-3: Completion Time Ratio of Scheduler and Lock Designs over Optimal (Rectangular Benchmark)

### Test 2: Comparing Scheduler to Optimal

Analysis of variance showed a significant main effect for type of designs (scheduler or optimal) ($F_{1,15} = 40.24$, $p = 1.3 \times 10^{-5}$) and number of tasks ($F_{13,15} = 8.90$, $p = 7.5 \times 10^{-5}$) on completion time. Multiple comparison analysis confirmed that the scheduler has longer completion time than optimal, which is expected. (For details, see appendix A, section A.2.)

Figure 5-4 plots the completion time of optimal design and the scheduler as a function of number of tasks. Completion time of the scheduler is on average 2.09 times longer than optimal ($mean = 2.0914$, $stderr = 0.0311$).

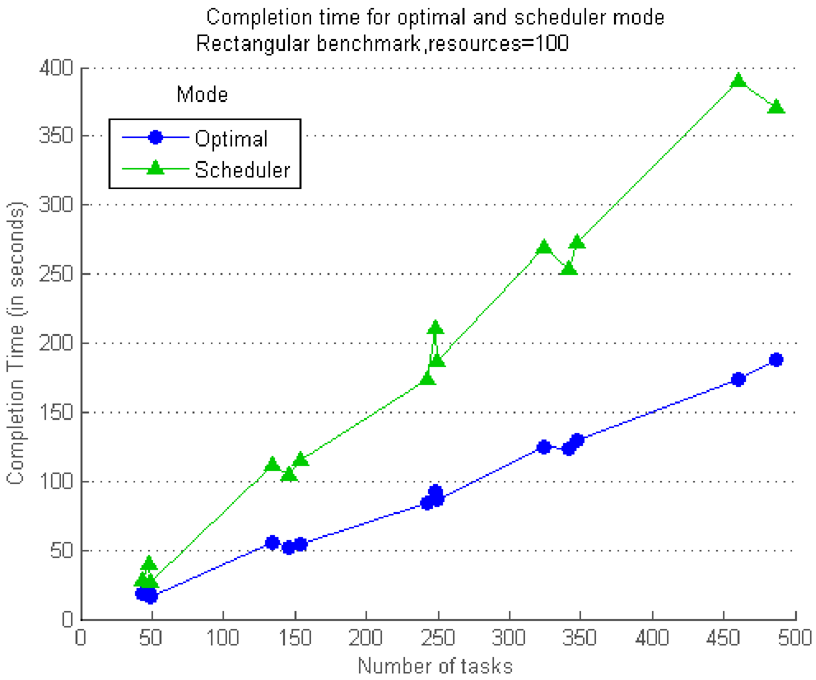


Figure 5-4: Completion Time for Optimal and Scheduler Design by Number of Tasks (Rectangular Benchmark). All data points in the data set are shown.

In summary, the test results suggest that scheduler performs better than simulated lock server in terms of minimizing completion time. It performs worse than optimal approximately with a factor of two on the rectangular benchmark.

## 5.3 Shared-Exclusive Hierarchical Benchmark

This benchmark tests the scheduling efficiency of the scheduler on a different kind of task sets. The tasks generated in this benchmark require both *shared* and *exclusive* ownerships. No optimal schedule is provided.

### 5.3.1 Benchmark

#### Task Generation

To generate tasks in this benchmark, resources are viewed as if they are organized in a tree (figure 5-5). Tasks randomly choose a resource (other than the root) and include in its resource requirement this resource's *exclusive* ownership and all its ancestors' *shared* ownership. A task can require more than one exclusive resources, in which case all of the ancestors of all the exclusive resources are also included in the task's resource requirement list, requiring *shared* ownership.

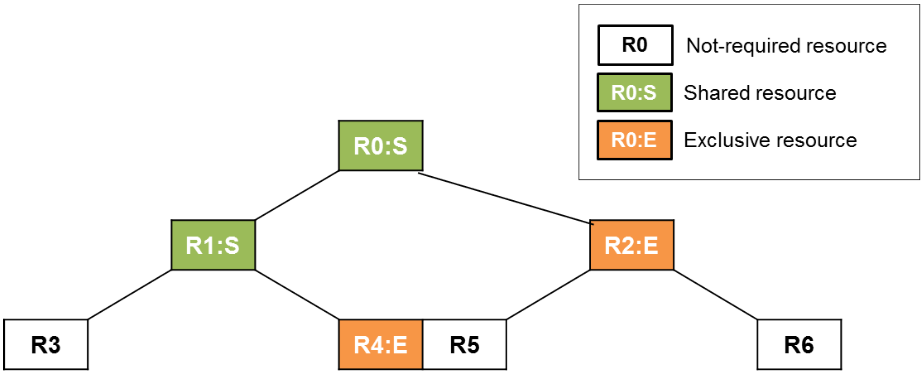


Figure 5-5: Example Task Resource Requirement in the Hierarchical Benchmark Benchmark Parameters: Number of levels = 3, branching factor = 2, maximum number of exclusive ownerships = 2.

Figure 5-5 shows an example of a task's resource requirement. The task requires *exclusive* ownership on resource R2 and R4 respectively. As a result, it also requires *shared* ownership on R4's ancestor, R1 and R0, and R2's ancestor, R0. The task's complete resource requirement is [R0:S, R1:S, R2:E, R4:E].

This benchmark was designed to resemble the resource organization in VIMS-managed virtual infrastructure. Virtual components in a virtual infrastructure are organized in a hierarchy. In a typical setup, at the top level there are data centers, which contain clusters. Clusters then contain hosts, which then contain virtual machines. API calls that API service receives usually requires *exclusive* lock on the specific virtual component that the operation is directed to *and shared* locks on all its containing components. The exclusive lock is to make sure that while this operation is making changes, no other operations can make changes on the same component at the same time. The shared locks on containers are used to allow operations on other components in the containers, while protecting the operation from disruptive changes on the container itself that *will* affect components it contains. For example, a `VirtualMachine.powerOn` request needs *exclusive* lock on the virtual machine itself and *shared* locks on the host, cluster and data center that contains it. This allows `powerOn` operations on other virtual machines to run in parallel, but protects the operation from running in parallel with container-exclusive operations such as `Host.shutdown`.

For this benchmark, the simulated lock server acquires resources in the same order as their indexes in the tree. (See numbers on the tree nodes in figure 5-5.) This is also meant to imitate the behavior of acquiring locks from the highest to the lowest in the locking hierarchy.

### Benchmark Parameters

The benchmark generates different task sets depending on the following parameters:

**Number of Levels:** Specifies the depth of the resource tree hierarchy.

**Branching Factor:** Specifies the number of children for each node in the hierarchy. The resource tree is a complete tree; each node except leaf nodes has the same number of children specified by the branching factor.

**Maximum Number of Exclusive Ownerships:** The maximum number of exclusive ownerships a task can acquire. For each task a random number between 1 and this number is used.

**Maximum Task Execution Time:** The longest running time of a task. For each task a random number between 1 and this number is used.

All tasks are queued at startup time.

### 5.3.2 Procedure

Table 5.3 shows values of parameters set in the tests, the independent variables chosen and the dependent variables that were measured. The number of levels and number of exclusive ownerships were chosen as an attempt to match the real setting in a VIMS-managed virtual infrastructure. But the branching factor in a real setting could be much larger than 10. (In particular, a typical cluster usually contains many more than 10 hosts.) 10 was chosen to accommodate to the hardware limitations of the testing machine. 3 tests were ran for each combination of type of design and number of tasks.

| **Benchmark Parameters** | |
|---|---|
| Number of levels | 4 |
| Branching factor | 10 |
| Maximum number of exclusive ownerships | 5 |
| Maximum task execution time | 20 |
| **Variables** | |
| Design | Scheduler, Lock |
| Number of tasks | 100, 300, 500, 700, 1000 |
| **Measures** | |
| Completion time (seconds) | Time spent for all tasks to finish execution |

Table 5.3: Experimental Setup for Hierarchical Benchmark Tests

### 5.3.3 Results

Analysis of variance showed significant main effects for type of design (lock or scheduler) ($F_{1,24} = 32.31$, $p < 1 \times 10^{-5}$) and number of tasks ($F_{4,24} = 29.58$, $p < 1 \times 10^{-8}$) on completion time. Multiple comparison analysis confirmed that the lock-based approach has longer completion time than the scheduler. (For details, see appendix A, section A.3, table A.5, A.6.)

Figure 5-6 plots the completion time of the scheduler and the lock-based approach as a function of number of tasks.

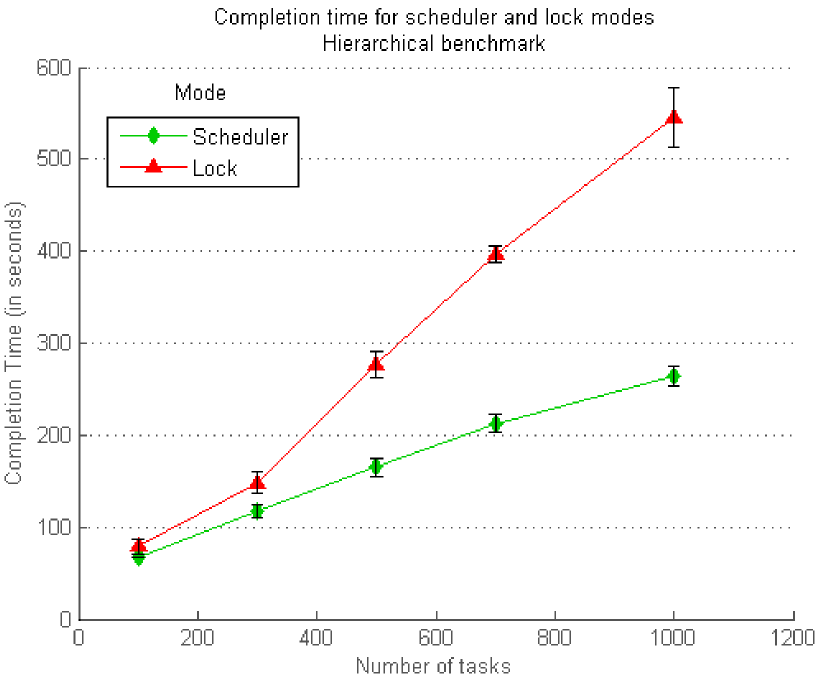


Figure 5-6: Completion Time for Scheduler and Lock Design by Number of Tasks (Hierarchical Benchmark)

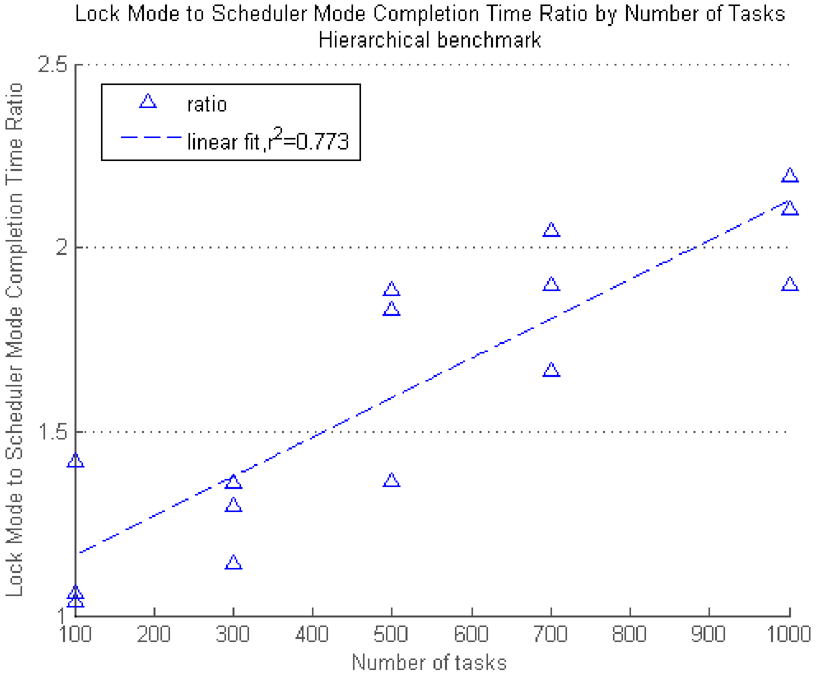


Figure 5-7: Completion Time Ratio of Scheduler to Lock-based Approach by Number of Tasks (Hierarchical Benchmark)

Figure 5-7 shows the completion time ratio of the lock-based approach to the scheduler. Completion time of the lock-based approach is on average 1.6 times longer than the scheduler ($mean = 1.6135$, $stderr = 0.1017$). Analysis of variance showed a significant main effect for number of tasks on the ratio ($F_{4,10} = 11.07$, $p = 0.0011$). Multiple comparison analysis showed a general trend of increasing ratio when the number of tasks increases. (For details, see appendix A, section A.3, table A.7, A.8.)

In summary, the results suggest that for the hierarchical benchmark, the simulated lock server again takes longer than the scheduler. And this difference tends to grow larger as the number of tasks increase.

## 5.4 Multiple-Instance Availability

The previous two benchmarks were run using one scheduler instance. This test ran multiple scheduler instances on a trivial benchmark and measures how completion time varies when instance crashes.

### 5.4.1 Benchmark

All tasks have the same resource requirement: *shared* ownership of all resources. In this way tasks do not conflict in resource usage. This excludes the influence of scheduling efficiency, and allows us to focus on the performance variations caused by instance crashes and restarts.

All tasks are queued at startup time. Task execution time is constant and the same for all tasks.

### 5.4.2 Procedure

Table 5.4 shows the setup of the test. 3 tests were run for each crash rate. All scheduler instances were run on the same machine.

Crash rate in this test refers to the number of crashes per second. For example, a crash rate of 0.1 means crashing once per 10 seconds. In this test in particular,

| **Benchmark Parameters** | |
|---|---|
| Task Execution Time (seconds) | 10 |
| Number of Resources | 10 |
| Number of Schedulers | 3 |
| Number of Tasks | 5000 |
| **Variables** | |
| Crash Rate (second$^{-1}$) | 0, 0.05, 0.08, 0.1, 0.2, 0.4, 0.6, 0.8, 1 |
| **Measures** | |
| Completion time (seconds) | Time spent for all tasks to finish execution |

Table 5.4: Experimental Setup for Multiple-Instance Availability Test

it means that one scheduler instance (chosen in a round-robin fashion among the 3 instances) is killed and immediately restarted every 10 seconds. When an instance is killed, its left-over tasks will be taken by other instances. When it restarts, it can start taking new tasks or tasks left over by other failed instances.

All tasks are queued at startup time.

### 5.4.3 Results

Analysis of variance showed a significant main effect for crash rate on completion time ($F_{8,21} = 22.9$, $p < 1 \times 10^{-7}$). Multiple comparison analysis showed a general trend of increasing completion time as crash rate increases, which is expected. (For details, see appendix A, section A.4.)

Figure 5-8 shows completion time as a function of crash rate. Linear regression analysis showed that completion time grows linearly with crash rate with r$^2$ above 0.8.

If we denote completion time without crash (i.e. crash rate $c = 0$) as $t_0$, and assuming that each scheduler instance's crash and restart takes constant time $t_c$, completion time $t$ at crash rate $c$ is

$$t = t_0 + t_c \times (t_0 \times c)$$

, which is a linear relationship between crash rate $c$ and completion time $t$. If we plug

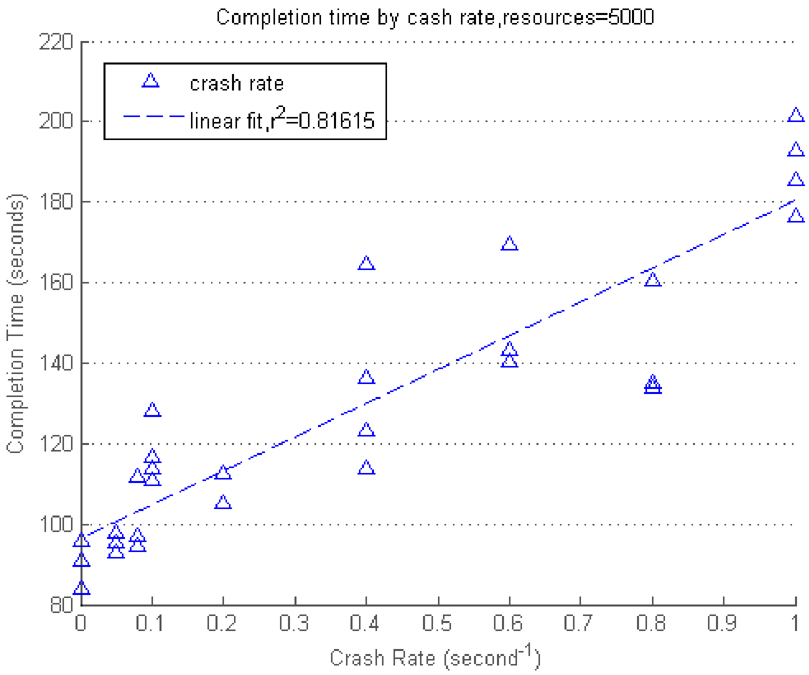


Figure 5-8: Completion Time by Crash Rate

in the values obtained from linear regression ($t = 85.03 \times c + 96.50$), we get a rough approximation of the crash-restart time $t_c = 881$ms ($t_0 = 96.50$s).

## 5.5 Multiple-Instance Scalability

In the following experiments, the same trivial benchmark (described in section 5.4.1) was used to run with varying number of scheduler instances and Zookeeper servers, in order to test how completion time is affected by those factors. Because all instances were run on the same machine, we are in fact measuring the scheduler and the Zookeeper server's overhead.

### 5.5.1 Procedure

Table 5.5 shows the experiment setup. All combinations of ZK and NS were run. NT=100 was run with all three different NRs; NT=1000 was run with NR=1 and 10; NT=10,000 was run with NR=1.

All scheduler and Zookeeper server instances were run on the same machine.

| | |
|---|---|
| **Benchmark Parameters** | |
| Task Execution Time (seconds) | 10 |
| **Variables** | |
| Number of Zookeeper Servers (ZK) | 1, 3, 5 |
| Number of Schedulers (NS) | 1, 3, 5 |
| Number of Tasks (NT) | 100, 1000, 10000 |
| Number of Resources (NR) | 1, 10, 100 |
| **Measures** | |
| Completion time (seconds) | Time spent for all tasks to finish execution |

Table 5.5: Experimental Setup for Multiple-Instance Tests

### 5.5.2 Results

Analysis of variance showed significant main effect for number of Zookeeper servers ($F_{2,153} = 24.87$, $p < 1 \times 10^{-9}$), number of tasks ($F_{2,153} = 313.13$, $p < 1 \times 10^{-54}$) and number of resources ($F_{2,153} = 3.77$, $p = 0.0253$) on completion time. Multiple comparison analysis showed that completion time increases with increasing number of Zookeeper servers, tasks and resources respectively. (For details, see appendix A, section A.5.)

Figure 5-9 plots completion time as a function of number of tasks for different numbers of Zookeeper servers. Completion time increases roughly linearly with the number of tasks. More Zookeeper servers will lead to longer completion time (steeper slopes in the graph).

There are two reasons that increasing the number of Zookeeper servers adds a noticeable overhead:

1. All Zookeeper updates, including node creation and deletion, need to go through the leader in the server quorum and then propagate to all servers [6]. As the number of servers increases, updates to Zookeeper may take longer.

2. Because all servers are run on one machine with two processors (for machine specifications see table 5.1), more than two server processes share computing power, and each process will take longer to respond.

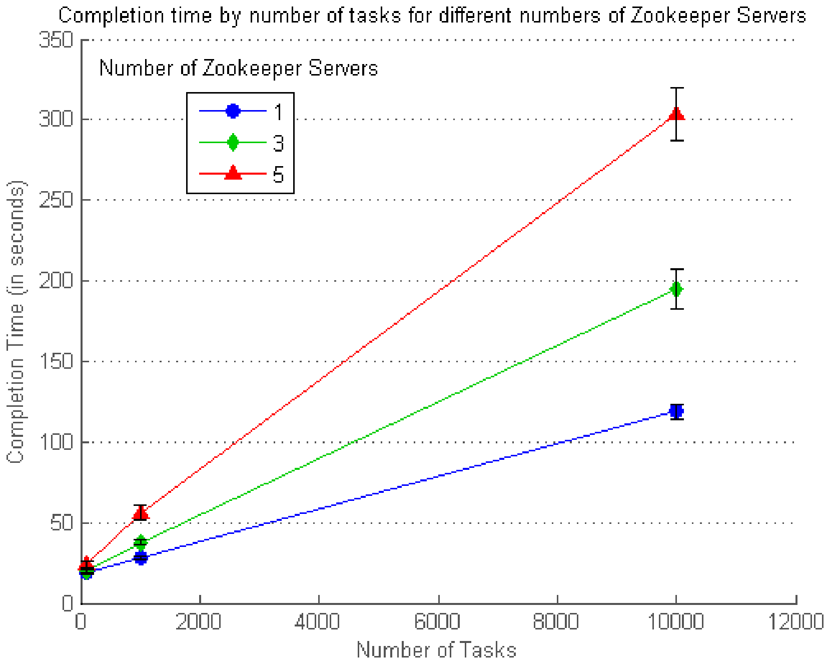


Figure 5-9: Completion Time by Number of Tasks for Different Numbers of Zookeeper Servers

Figure 5-10 plots completion time as a function of the number of tasks for different number of schedulers. Variations on completion time for different numbers of schedulers are not distinct according to the data points because of their overlapping error bars. This shows that scheduler's overhead is not large, which is expected because instances are lightweight.

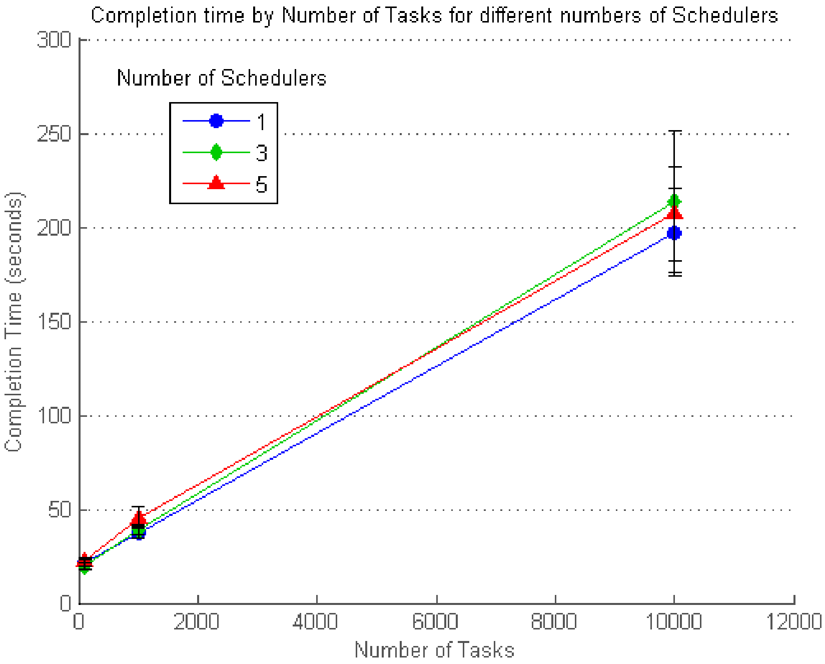


Figure 5-10: Completion Time by Number of Tasks for Different Numbers of Schedulers

# Chapter 6

# Conclusion

This chapter first summarizes contributions made in this thesis, and then suggests some future work.

## 6.1 Contributions

This thesis proposed a design and implementation of a distributed real-time scheduler for the virtual infrastructure management system (VIMS) that replaced the need of a lock server. A round-robin scheduling algorithm that orders tasks based on their needs of locks was proposed.

A distributed design of the scheduler based on the proposed algorithm was made that relies on Zookeeper as the scheduler instances' shared central data service that maintains the global task queue and resource availability tree. A set of Zookeeper-based scheduling procedures were implemented by making careful use of Zookeeper's internal consistency guarantees to ensure correctness of the shared data, while keeping as much concurrency on data structure accesses as possible. The scheduler design was also made highly available by implementing a simple failover scheme that relies on persisted task states on Zookeeper.

The scheduler's efficiency with respect to task completion time was tested and compared with optimal schedule and lock server. The scheduler is shown to have significantly shorter completion time than the simulated lock server (the scheduler takes about one half of lock server's completion time), and its completion time is about two times of the optimal completion time on both the exclusive-only rectangular benchmark and the shared-exclusive hierarchical benchmark. The scheduler's availability was tested against varying crash rates. Experimental results have shown that the scheduler survives crashes well and its performance degrades roughly linearly with increasing crash rates. The system overhead was also tested by starting multiple scheduler instances and Zookeeper servers. The results suggest that the scheduler is actually lightweight—performances under different numbers of scheduler instances don't show much difference.

## 6.2 Future Work

This section mentions future work, some of which are about new features that could be useful to add, others are related to improving the current design.

### 6.2.1 Task Starvation

The current design of the scheduler does not make special efforts to keep fairness among tasks. Starvation is a potential problem in the current design, i.e. some tasks may have to wait for a long time without getting its required resources. For example, if the task requires a large number of resources, while many small tasks only require a few, the scheduler may keep succeeding in getting resources for the smaller tasks and ignoring the task that requires many more resources because its requirement is hard to satisfy. To solve this problem, some form of a timeout mechanism may be implemented so that a task that is waiting for too long needs to have the priority of getting resources.

### 6.2.2 Task Priority

The current design of the scheduler does not have a notion of task priorities. Task priority may be desired if some tasks are important and should be dispatched as soon as possible. It may also be useful for solving the starvation problem, by assigning starved tasks higher priorities.

### 6.2.3 Asynchronous Zookeeper Operations

The current scheduler makes Zookeeper operations through synchronous calls, i.e. the call will only return when the operation is carried out on Zookeeper servers. Zookeeper has another set of asynchronous calls that use callbacks when operations finish execution. For asynchronous calls the initial call from scheduler returns immediately without waiting for a reply. Various performance tests have shown that Zookeeper asynchronous operations have significantly larger throughput than synchronous calls [6, 8, 17]. For example, creating a thousand nodes on Zookeeper by making asynchronous calls finishes much faster than using synchronous calls. It may improve performance if the scheduler can use asynchronous calls for interacting with Zookeeper.

### 6.2.4 Resource Granularity

We have described that one virtual component is a resource, but the resources could be further divided up if it allows more concurrency. For example, task `VirtualMachine.powerOn` could be run in parallel with `VirtualMachine.rename`, even though they are both performed on the same virtual component.

### 6.2.5 Estimating Task Execution Time

Several algorithms mentioned in chapter 2 requires to know the tasks' execution time. The scheduler can potentially perform more sophisticated scheduling algorithms if it has some idea about the running time of the tasks. VIMS does not currently provide estimations of tasks' execution time because it depends on many factors including the target component's current load, memory, CPU and disk usages. But it is not entirely impossible to estimate the tasks' execution time. If the scheduler's efficiency can be greatly improved with known task execution times, it might be worthwhile to make these estimations.

### 6.2.6 A Priori Task Resource Requirements Feasibility

The scheduler requires a priori knowledge of tasks' resource needs before task execution. Resource requirements are currently specified by a descriptive language as part of the API specifications, and resource IDs are dynamically resolved by the API service querying the database. Further evaluations need to be done to identify the cases in which this is not readily achievable. If some task cannot know what resource it needs until it has run part of its procedure, then a priori knowledge may not be attainable. For example, a migration of virtual machines from one host to another where the destination host is dynamically determined based on host server loads might not know the destination host until it has run the load inspection procedure.

One potential solution may be to explore the possibility of splitting the tasks whose resources cannot be resolved up front into subtasks that can acquire resources separately. A later subtask may use the results from the previous subtasks for resolving its own resource needs. Another potential way to deal with this is to change the scheduling algorithm so that it allows tasks to acquire additional resources after they have been dispatched.

# Appendix A

# Statistical Tests

## A.1 Rectangular Benchmark Test 1

Table A.1: Rectangular Benchmark Test 1: Two-way Anova on Completion Time

| Source | Sum Sq. | d.f. | Mean Sq. | F | Prob>F |
|---|---|---|---|---|---|
| Mode | 1.0740e+005 | 2 | 5.3700e+004 | 13.3776 | 3.2364e-004 |
| Number of Tasks | 9.5858e+004 | 7 | 1.3694e+004 | 3.4114 | 0.0183 |
| Error | 6.8241e+004 | 17 | 4.0142e+003 | | |
| Total | 2.7150e+005 | 26 | | | |

Table A.2: Rectangular Benchmark Test 1: Tukey HSD for Mode on Completion Time 95% confidence interval

| Group1 | Group2 | Diff. Mean | Lower | Upper |
|---|---|---|---|---|
| Optimal | Scheduler | -40.9539 | -117.5734 | 35.6656 |
| Optimal | Lock | -149.4812 | -226.1008 | -72.8617 |
| Scheduler | Lock | -108.5273 | -31.9078 | |

## A.2 Rectangular Benchmark Test 2

Table A.3: Rectangular Benchmark Test 2: Two-way Anova on Completion Time

| Source | Sum Sq. | d.f. | Mean Sq. | F | Prob>F |
|---|---|---|---|---|---|
| Mode | 9.0794e+004 | 1 | 9.0794e+004 | 40.2450 | 1.3178e-005 |
| Number of Tasks | 2.6096e+005 | 13 | 2.0074e+004 | 8.8980 | 7.6390e-005 |
| Error | 3.3841e+004 | 15 | 2.2560e+003 | | |
| Total | 3.8560e+005 | 29 | | | |

Table A.4: Rectangular Benchmark Test 2: Tukey HSD for Mode on Completion Time 95% confidence interval

| Group1 | Group2 | Diff. Mean | Lower | Upper |
|---|---|---|---|---|
| Optimal | Scheduler | -110.0269 | -146.9942 | -73.0596 |

## A.3 Hierarchical Benchmark Test

Table A.5: Hierarchical Benchmark Test: Two-way Anova on Completion Time

| Source | Sum Sq. | d.f. | Mean Sq. | F | Prob>F |
|---|---|---|---|---|---|
| Mode | 1.1500e+005 | 1 | 1.1500e+005 | 32.3118 | 7.4404e-006 |
| Number of Tasks | 4.2107e+005 | 4 | 1.0527e+005 | 29.5784 | 5.8082e-009 |
| Error | 8.5415e+004 | 24 | 3.5589e+003 | | |
| Total | 6.2148e+005 | 29 | | | |

Table A.6: Hierarchical Benchmark Test: Tukey HSD for Mode on Completion Time 95% confidence interval

| Group1 | Group2 | Diff. Mean | Lower | Upper |
|---|---|---|---|---|
| Scheduler | Lock | -123.8255 | -168.7847 | -78.8664 |

Table A.7: Hierarchical Benchmark Test: One-way Anova on Lock/Scheduler Ratio

| Source | Sum Sq. | d.f. | Mean Sq. | F | Prob>F |
|---|---|---|---|---|---|
| Number of Tasks | 1.7717 | 4 | 0.4429 | 11.0696 | 0.0011 |
| Error | 0.4001 | 10 | 0.0400 | | |
| Total | 2.1718 | 14 | | | |

Table A.8: Hierarchical Benchmark Test: Tukey HSD for Number of Tasks (NT) on Lock/Scheduler Ratio
95% confidence interval

| Group1 | Group2 | Lower | Diff. Mean | Upper |
|---|---|---|---|---|
| NT=100 | NT=200 | -0.6316 | -0.0941 | 0.4434 |
| NT=100 | NT=300 | -1.0592 | -0.5217 | 0.0159 |
| NT=100 | NT=400 | -1.2352 | -0.6977 | -0.1602 |
| NT=100 | NT=500 | -1.4294 | -0.8919 | -0.3544 |
| NT=200 | NT=300 | -0.9651 | -0.4276 | 0.1100 |
| NT=200 | NT=400 | -1.1411 | -0.6036 | -0.0661 |
| NT=200 | NT=500 | -1.3353 | -0.7978 | -0.2603 |
| NT=300 | NT=400 | -0.7136 | -0.1761 | 0.3615 |
| NT=300 | NT=500 | -0.9078 | -0.3703 | 0.1673 |
| NT=400 | NT=500 | -0.7317 | -0.1942 | 0.3433 |

## A.4 Multiple-Instance Availability Test

Table A.9: Multiple-Instance Availability Test: One-way Anova for Crash Rate on Completion Time

| Source | Sum Sq. | d.f. | Mean Sq. | F | Prob>F |
|---|---|---|---|---|---|
| Mode | 2.8355e+004 | 8 | 3.5444e+003 | 22.8998 | 1.0214e-008 |
| Error | 3.2504e+003 | 21 | 154.7789 | | |
| Total | 3.1606e+004 | 29 | | | |

Table A.10: Multiple-Instance Availability Test: Tukey HSD for Crash Rate (CR) on Completion Time
95% confidence interval

| Group1 | Group2 | Diff. Mean | Lower | Upper |
|---|---|---|---|---|
| CR=0 | CR=0.05 | -40.0135 | -5.0337 | 29.9462 |
| CR=0 | CR=0.08 | -45.8992 | -10.9193 | 24.0605 |
| CR=0 | CR=0.1 | -59.8097 | -27.0891 | 5.6316 |
| CR=0 | CR=0.2 | -54.7175 | -19.7377 | 15.2422 |
| CR=0 | CR=0.4 | -76.9660 | -44.2453 | -11.5247 |
| CR=0 | CR=0.6 | -95.7329 | -60.7530 | -25.7731 |
| CR=0 | CR=0.8 | -87.7799 | -52.8000 | -17.8201 |
| CR=0 | CR=1 | -131.3937 | -98.6731 | -65.9524 |
| CR=0.05 | CR=0.08 | -40.8655 | -5.8857 | 29.0942 |
| CR=0.05 | CR=0.1 | -54.7761 | -22.0554 | 10.6652 |
| CR=0.05 | CR=0.2 | -49.6839 | -14.7040 | 20.2759 |
| CR=0.05 | CR=0.4 | -71.9323 | -39.2117 | -6.4910 |
| CR=0.05 | CR=0.6 | -90.6992 | -55.7193 | -20.7395 |
| CR=0.05 | CR=0.8 | -82.7462 | -47.7663 | -12.7865 |
| CR=0.08 | CR=1 | -126.3601 | -93.6394 | -60.9188 |
| CR=0.08 | CR=0.1 | -48.8904 | -16.1697 | 16.5509 |
| CR=0.08 | CR=0.2 | -43.7982 | -8.8183 | 26.1615 |
| CR=0.08 | CR=0.4 | -66.0467 | -33.3260 | -0.6053 |
| CR=0.08 | CR=0.6 | -84.8135 | -49.8337 | -14.8538 |
| CR=0.08 | CR=0.8 | -76.8605 | -41.8807 | -6.9008 |
| CR=0.08 | CR=1 | -120.4744 | -87.7537 | -55.0331 |

| Group1 | Group2 | Diff. Mean | Lower | Upper |
|---|---|---|---|---|
| CR=0.1 | CR=0.2 | -25.3692 | 7.3514 | 40.0721 |
| CR=0.1 | CR=0.4 | -47.4497 | -17.1563 | 13.1372 |
| CR=0.1 | CR=0.5 | -66.3846 | -33.6639 | -0.9433 |
| CR=0.1 | CR=0.8 | -58.4316 | -25.7109 | 7.0097 |
| CR=0.1 | CR=1 | -101.8774 | -71.5840 | -41.2906 |
| CR=0.2 | CR=0.4 | -57.2283 | -24.5077 | 8.2130 |
| CR=0.2 | CR=0.6 | -75.9952 | -41.0153 | -6.0355 |
| CR=0.2 | CR=0.8 | -68.0422 | -33.0623 | 1.9175 |
| CR=0.2 | CR=1 | -111.6561 | -78.9354 | -46.2148 |
| CR=0.4 | CR=0.6 | -49.2283 | -16.5077 | 16.2130 |
| CR=0.4 | CR=0.8 | -41.2753 | -8.5547 | 24.1660 |
| CR=0.4 | CR=1 | -84.7212 | -54.4277 | -24.1343 |
| CR=0.6 | CR=0.8 | -27.0269 | 7.9530 | 42.9329 |
| CR=0.6 | CR=1 | -70.6407 | -37.9201 | -5.1994 |
| CR=0.8 | CR=1 | -78.5937 | -45.8731 | -13.1524 |

## A.5 Multiple-Instance Scalability Test

Table A.11: Multiple-Instance Test: Four-way Anova on Completion Time

| Source | Sum Sq. | d.f. | Mean Sq. | F | Prob>F |
|---|---|---|---|---|---|
| Number of Zookeeper Servers | 4.9022e+004 | 2 | 2.4511e+004 | 24.8669 | 4.4571e-010 |
| Number of Schedulers | 571.8491 | 2 | 285.9245 | 0.2901 | 0.7486 |
| Number of Tasks | 6.1730e+005 | 2 | 3.0865e+005 | 313.1312 | 8.2242e-055 |
| Number of Resources | 7.4233e+003 | 2 | 3.7116e+003 | 3.7655 | 0.0253 |
| Error | 1.5081e+005 | 153 | 985.6844 | | |
| Total | 9.2520e+005 | 161 | | | |

Table A.12: Multiple-Instance Test: Tukey HSD for Number of Zookeeper Servers (ZK) on Completion Time
95% confidence interval

| Group1 | Group2 | Lower | Diff. Mean | Upper |
|---|---|---|---|---|
| ZK=1 | ZK=3 | -30.2935 | -16.1327 | -1.9719 |
| ZK=1 | ZK=5 | -56.3815 | -42.2207 | -28.0598 |
| ZK=3 | ZK=5 | -40.2488 | -26.0880 | -11.9271 |

Table A.13: Multiple-Instance Test: Tukey HSD for Number of Tasks (NT) on Completion Time
95% confidence interval

| Group1 | Group2 | Lower | Diff. Mean | Upper |
|---|---|---|---|---|
| NT=100 | NT=1,000 | -39.2911 | -25.1303 | -10.9695 |
| NT=100 | NT=10,000 | -213.9097 | -195.1767 | -176.4437 |
| NT=1,000 | NT=10,000 | -188.7794 | -170.0464 | -151.3134 |

Table A.14: Multiple-Instance Test: Tukey HSD for Number of Resources (NR) on Completion Time
95% confidence interval

| Group1 | Group2 | Lower | Diff. Mean | Upper |
|---|---|---|---|---|
| NR=1 | NR=10 | -23.8031 | -9.6422 | 4.5186 |
| NR=1 | NR=100 | -40.0752 | -21.3421 | -2.6091 |
| NR=10 | NR=100 | -30.4330 | -11.6999 | 7.0331 |

# Bibliography

[1] Luca Becchetti and Stefano Leonardi. Nonclairvoyant scheduling to minimize the total flow time on single and parallel machines. *Journal of the ACM*, 51(4):94–103, 2001.

[2] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[3] E G Coffman and R L Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1(3):200–213, 1972.

[4] M H Eich. Graph directed locking. *Software Engineering IEEE Transactions on*, 14(2):133–140, 1988.

[5] T C Hu. Parallel Sequencing and Assembly Line Problems. *Operations Research*, 9(6):841–848, 1961.

[6] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. ZooKeeper : Wait-free coordination for Internet-scale systems. *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, 8:11–11, 2010.

[7] Zvi Kedem and Abraham Silberschatz. Controlling concurrency using locking protocols, 1979.

[8] N Keywal. zookeeper-user - sync vs. async vs. multi performances, 2012.

[9] Joseph Y-T Leung. 03. Some Basic Scheduling Algorithms. In *Handbook of Scheduling Algorithms Models and Performance Analysis*, volume 37, chapter 3, pages 1221–1222. Chapman and Hall/CRC, 2004.

[10] Joseph Y-t Leung. *Handbook of SCHEDULING Algorithms, Models, and Performance Analysis*, volume 128 of *Computer and Information Sciences Series*. CRC Press, 2004.

[11] J M Moore. An n Job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs. *Management Science*, 15(1):102–109, 1968.

[12] Kirk Pruhs, Jiri Sgall, and Eric Torng. Online Scheduling. In J Leung, editor, *Handbook of Scheduling Algorithms Models and Performance Analysis*, chapter 15, pages 196–231. CRC Press, 2004.

[13] Jir Sgall. On-Line Scheduling — A Survey. *LNCS 1442*, pages 192–231, 1998.

[14] A Silberschatz and Z M Kedam. A Family of Locking Protocols for Database Systems that are Modeled by Directed Graphs. *IEEE Transactions on Software Engineering*, SE-8(6):558–562, 1982.

[15] Abraham Silberschatz and Zvi Kedem. Consistency in Hierarchical Database Systems. *Journal of the ACM*, 27(1):72–80, 1980.

[16] Apache Zookeeper. ZooKeeper (ZooKeeper 3.4.3 API).

[17] Apache Zookeeper. ZooKeeper/ServiceLatencyOverview - Hadoop Wiki.

[18] Apache Zookeeper. ZooKeeper Programmer's Guide, 2012.