A Service Migration Case Study: Migrating the Condor Schedd

Joe Meehean and Miron Livny
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706
{jmeehean|miron}@cs.wisc.edu

March 11, 2005

Abstract

Service migration has become an important topic due the rising interest in both service-based architectures and mobile computing. We have identified two core problems associated with migrating a service: packaging the service binaries and data in a fashion that allows it to be restarted at a remote site and locating a service after it has migrated. Many implementations of service migration assume homogeneous host architectures as well as uniform file access. Additionally, some implementations require that migration occur in kernel-space. We require that a service capture its own state using configuration files and operation logs. This state is then marshalled to be machine architecture independent as well as independent of any file system or mount point. We call this technique *service-defined logical checkpointing*, it occurs entirely in user-space and significantly eases the migration of a service. We mobilized the Condor High Throughput System's distributed scheduling service (schedd) to illustrate the use of service-defined logical checkpointing to migrate a service. Further, we created a specialized Condor command and associated Condor job that can be used to migrate a schedd to a specific host or to a host matching an arbitrary set of requirements, including CPU load.

1 Introduction

Services and service-based architectures are becoming an increasingly popular way to implement distributed systems. Web developers are creating applications using the distributed web service approach offered by J2EE [4] and .Net [6] solutions. The Open Grid Services Architecture details an extension of web services for the grid, in which grid functionality is provided by a series of transient and stateful grid services [14]. Several operating systems are implemented using a local service based architecture, including Kea [30] and ExoKernel [13]. In addition, the recent popularity of P2P content distribution networks, such as Kazaa [5], has caused many users to install peer-based services on their desktop machines.

This new reliance on services has created a need for a more robust brand of service; services must be mobilized. Server-side service mobility may be used as a load balancing technique. Generally a service is bound to a particular server or cluster of servers. Increased demand for a given service may cause a poor performance on specific server while within the same server room other machines remain relatively idle. Additionally, it is occasionally necessary to take a server offline for maintenance. Service mobility would allow a given service to migrate to another machine while maintenance occurs and migrate back when maintenance is finished. Client-side service mobility allows more freedom for an increasingly mobile computing population. A user may wish to migrate a service from their desktop to a laptop for roaming, or to migrate a service from a remote desktop to a local desktop. Or more dynamically a user may wish to run a service, but does not want the service to interfere with interactive tasks the user is running. A service could be migrated from active desktops to inactive desktops throughout the course of the day.

We have identified problems associated with migrating a service and have applied a series of techniques to solve them in user-space without a uniform view of the file system. Further, we have mobilized the distributed scheduling service of the Condor High Throughput Computing System to test the effectiveness of our approach.

In the remainder of this paper, we refer to the machine a service is migrating from as the *source* machine and the machine a service is migrating to as the *target* machine. In Section 2, we discuss our basic designs goals. Section 3 provides an overview of our service migration techniques. We provide a brief overview of the Condor High Throughput Computing System in Section 4. In Section 5, we delve into the implementation details of mobilizing the distributed scheduler. Section 6 provides performance metrics of the mobilized service. We briefly discuss related work in Section 7 and lay out our future work and conclusions in Section 8.

2 Design Goals

At the outset of this project we set forth a series of design goals, restrictions, and assumptions for completing service migration. Many of these goals are influenced by the Condor High Throughput Computing System [29] because in the Condor system homogeneity can rarely be assumed. Our basic design goals are as follows.

- A service should behave in the same manner after migrating.
- A mobile service does not require a specialized compiler.
- Migrating a service will occur entirely in user-space.
- A service wishing to be mobilized may be modified to ease capturing its state.
- A migrated service will have no residual dependencies on the source site. This means that there will be no need for contact between the migrated service and its previous host after migration.
- A migrated service cannot assume uniform access to a distributed file system.
- The data migrated with a service cannot assume the same operating system or architecture.
- It is acceptable for the service to be temporarily unavailable during migration.

Several of these goals may seem arbitrary or conflicting so we attempt to justify our design choices. We anticipated that modifying a service is easier than modifying all of the operating systems the service may wish to migrate to. Further, performing migration in kernel-space may make a migration framework fragile and susceptible to breakdowns between versions of the same kernel let alone other operating systems [12]. Additionally, we felt that attempting to capture the state of a service using a specialized compiler would limit our ability to migrate services written in non type-safe languages such as C/C++ [26]. While distributed file systems are popular we felt it would be naive to assume that a migrating service would have homogeneous access to its home file system at every host. A service's home file system may be unavailable or mounted differently at the target site. The theme of our goals is heterogeneity and with that in mind we deemed that a service's data should be migrated in an operating system and host architecture independent manner. A service may be ported to several different operating systems and architectures. Therefore, when migrating a service we cannot assume that the service's data will be interpreted the same way on the target site. Finally, we allow the user to notice a brief interruption in a migrating service. Adding service redirection at both the network and application layers may allow us avoid this interruption but would violate our design goal of having no residual dependencies.

3 Architecture

We have identified two core problems associated with migrating a service: physically moving the service binaries and data in a manner which allows for execution at a remote site (Section 3.1) and locating the service once it has migrated (Section 3.2).

3.1 Service-Defined Logical Checkpointing

Physical process checkpointing would allow us to suspend and resume a service, but violates many of our basic design goals. Some implementations of physical checkpointing require that the checkpointing occur in kernel-space [12, 10, 20], while others require remote kernel calls, which violates our goal of removing residual dependencies [16]. Virtual machine solutions checkpoint a guest operating system's file system as well as its processes, but require that a service always run in a virtual machine [15]. This requires either significant slowdown or the assumption that a service will never be migrated to a machine with a different architecture. The Tui System [26] provides a mechanism to translate a process's data into an architecture independent intermediate form, but requires a specialized compiler and works only for type-safe programs.

Since our design goals allow changes to a service we have decided that a service should be able to contribute to checkpointing. We define our checkpointing techniques as *service-defined logical checkpointing*. Essentially, the service captures its own state in a service-dependent manner. This state must be translated into an intermediate form breaking all dependencies on uniform file access, kernel version, and architecture. Finally, this marshalled state must be packaged along with any files needed by the service into a single checkpoint file.

3.1.1 Mobile Service State Capture

A service's state is divided into static state and dynamic state. The static state of service are values which are not likely to change over the run time of the service. Static state could include values like the service's name, logging level, and the location of helper programs. In contrast, a service's dynamic state are values which change frequently, as frequently as several times a second. Dynamic state could include values like the number of requests currently being serviced, a list of open files, and the state of current requests. Static state can be thought of as read-only while dynamic state can be read and written. Due to their different nature, it makes sense to capture these states differently. Static state can be effectively captured even before a service begins executing, while dynamic state must be captured in real-time.

Static service state can be captured in configuration or properties files. Configuration files allow users to customize applications or libraries for their specific needs, and for this reason many applications and code libraries already use configuration files. These files range in size and complexity from the dozens of logically linked files needed by Apache Struts [1] to the relatively simple configuration file for Emacs [2].

Dynamic service state can be captured using logging or journaling. Logging is the act of writing events in the order they occur to non-volatile storage. Typically logging is done along-side actual processing and stored in a separate file to provide debugging information and failure recovery. Write-ahead logging is used by databases to provide transactional logic and failure recovery [23]. The Log-Structured file system (LFS) differs from database transaction logging in that the actual file system data is stored in a log format [25]. Using both LFS and database logs as a model, a service can export changes to its dynamic state as updates to an operations log file. After migration, a service need only replay its log to reconstruct its dynamic state.

3.1.2 Marshalling Service State

Once a service's state is captured it must be migrated with the service. For performance and efficiency a service's state may contain assumptions about the environment it is executing in. These assumptions include things like the location of libraries, the architecture of the host machine, the mount point of a distributed file system, and the location of service-specific files and directories. These assumptions must be removed from a service's state to produce a host-independent checkpoint. Therefore, a service's state must be marshalled prior to checkpointing. The details of marshalling are service dependent, however, the basic principles are the same.

Base data types, integers, floating-point numbers, and raw bytes, must be modified into a machine agnostic representation. Flat data types, a series of base data types maintained in a structure or array, must be modified in such a way that the correct order can be determined at the remote host. Complex data types are structures that may contain pointers or references to other structures which must also be marshalled. There are a wide variety of techniques for marshalling base, flat, and complex data types [22]. Choosing the appropriate technique is service dependent and relies heavily on how the service's state is captured.

If we do not assume uniform file access between migration points then file paths require a specialized marshalling scheme. When discussing file paths we are referring to absolute file paths and believe this simplification is legitimate since a relative file path can be easily converted to an absolute file path. One might consider a file path a flat structure since it is composed of an array of characters. However, in service migration a file path is more closely related to a complex data type in that special care must be taken not to lose important context. The path portion of a file name provides context for the file, if it is located in a `bin` directory then the file is likely an executable binary. Further, some of this context is only relevant at a particular host. For example the path `/unsup/vdt/globus/` may be the location of the Globus program files at host A, but this path may be meaningless on host B. Relations between files may be inferred by their paths names, but without extremely intelligent software the overall context is lost. For example, the files `kbattleship` and `kasteroids` may both reside in `/s/kde/bin/` which implies a relationship, but it would be difficult for a program interpreting this relationship to recognize that this path represents the binary executables for KDE version 3.2. To prevent losing this context and to decouple a file's path from its current machine we can replace portions of a path name with a macro. The path `/unsup/vdt/globus/` could be replaced with `$GLOBUS/`, or the path `/s/kde/bin/kbattleship` can be modified to `$KDE_3_2/bin/kbattleship`. Many application already use a similar approach to locate needed libraries. For example, the Apache Jakarta Tomcat application server uses an environment variable, `JAVA_HOME`, to determine the location of the Java Runtime Environment [8].

Demarshalling these path names requires that a host have the appropriate macros defined. These macro definitions encapsulate important information about a host, such as the location of code libraries and helper binaries. It would be very difficult to migrate a service to a machine that the service knew absolutely nothing about. These macros can be considered an addition to the minimal set of information needed to demarshall a service at an arbitrary host. Other items in this minimal set include operating system version and machine architecture. Further, decoupling files from their absolute paths allows us to migrate these files with a service when they are not already stored at the target site. The file migration implementation can set the macros at the remote site to point to the location of the newly migrated files. Even if a file is stored in a distributed file system and reachable from a target site, the distributed file system may only be accessible via a high latency connection making file migration to a local file system a performance improvement.

3.2 Mobile Service Location

A common solution to general service location is to employ a naming service that maps a persistently named service instance to a transient IP address and port number. DNS [19], a classic example, maps dot-separated hierarchical names to IP addresses for email and WWW. Another example is Sun's Port Mapper [18] which maps a service name to the port number on which the service is currently accepting requests.

Although naming services provide a solution to general service location, mobile services introduce an added difficulty by potential changing their IP and port number frequently. A naming service's effectiveness in providing a location for a mobile service is limited by its latency in updating its service mappings. In addition, clients of a mobile service must be prepared to reconsult the naming service any time a connection fails.

4 Condor

We mobilized the distributed scheduling service of the Condor High Throughput System to test using our techniques for actual service migration. A basic knowledge of Condor is necessary to understand our implementation of a mobilized scheduling agent. This section provides a brief overview of Condor.

4.1 Condor Architecture

The Condor High Throughput Computing System is composed of a collection of machines loosely coupled into a pool. The machines in a pool work together to collectively provide high throughput processing, measuring work accomplished in hours not milliseconds. Each pool must have at least one machine representing each of the following roles, see Figure 1. A *submit* machine accepts job submissions from users and must run the *schedd*, the Condor distributed scheduling service. An *execute* machine runs jobs submitted to Condor and must be running the *startd*, the Condor distributed computation service. A startd is the representative of the machine's owner, enforcing the owner's policy regarding when jobs

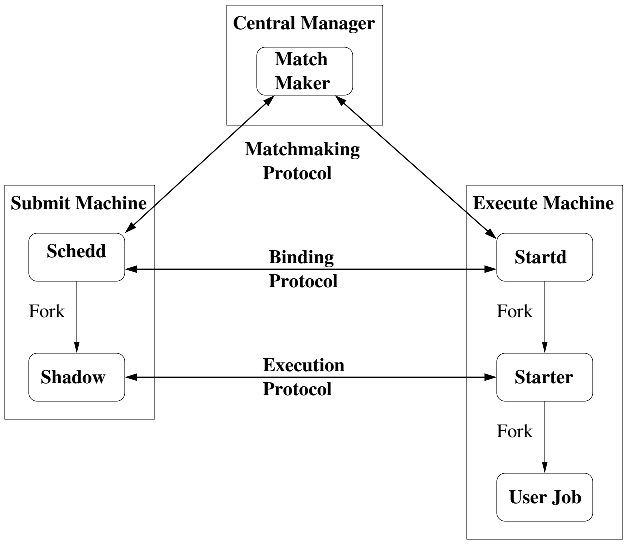

Figure 1: Machine Roles in a Condor Pool

may be run on the machine, who can run jobs on the machine, and which jobs to give priority to. The third and final role in the Condor pool is the central *match maker*. The match maker matches jobs submitted to the schedds with machine resources represented by the startds. A machine may serve many roles and there can be many submit and execute machines, but at the present there is only one match maker per pool [29].

Because a Condor pool may consist of machines with different operating systems and architectures the match making language must be machine independent. Condor uses the ClassAds [24] language, which was inspired by classified advertisements, for describing both jobs and resources. ClassAds are composed of a series of attribute-expression pairs, e.g., `OpSys = LINUX`. The schedd converts a job submission description into a job class ad, stores it internally, then transmits it to the match maker for matching. On the execute machine, the startd converts the machine specifications and the owner's policy into a machine class ad and also submits it the match maker. The match maker in turn attempts to match job ads to machine ads. When the match maker finds a possible match it notifies each party, indicating that a match has occurred. However, the final binding between job and machine is negotiated by the schedd and startd, if an agreement can't be reached a match is not made. After binding has taken place the schedd forks a process called the *shadow* which is responsible for staging the job to the remote execution site as well as providing remote access to any files the job may require. On the execute machine the startd forks a process called the *starter* which is responsible for setting up the execution environment, forking the job, and monitoring the job's execution.

4.2 Advantages of using the Schedd as an Example Implementation

The schedd is a good example for illustrating service migration using our techniques because it has several desirable features of a complex service. The schedd maintains several

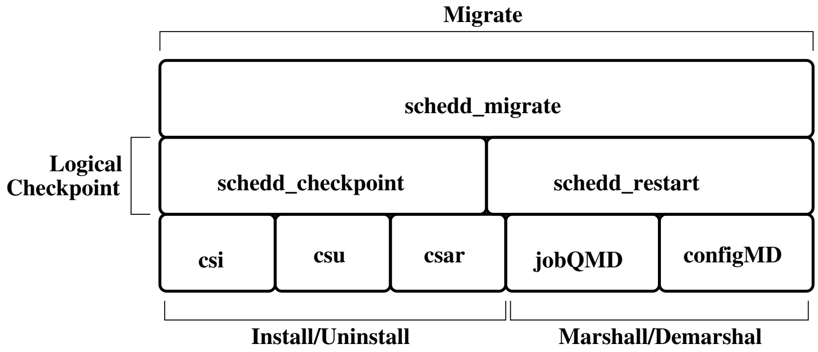

Figure 2: Migration Infrastructure

files related to a job's execution. These files must be migrated with the schedd allowing us to test our path marshalling. Additionally, the schedd forks other processes, meaning the process binaries must be migrated and their remote paths determined. Further, the schedd already captures its state using an operation log and a configuration file. Basic and flat data types are already in a machine independent form because this log file is written in a marked up version of ClassAds. The schedd already logging as a failure recovery mechanism because it is designed to work in a dynamic environment where resources may crash or be reclaimed. Additionally, due to the heterogeneity of resources within a Condor pool, which can span several networks and administrative domains, the schedd is dynamically configurable using both configuration files and environment variables. Also, a schedd periodically sends a portion of its state to the match maker, including its IP address and port number. When a service or user wishes to locate a specific schedd it queries the match maker. There is only one matchmaker per pool so their is no added latency involved in updating replicas, which satisfies our requirement for a low-latency naming service.

5 Implementation

Naming is handled by the match maker, and state capture is handled by a combination of the schedd's operation log, *jobQ*, and schedd's configuration file, *condor_config*. Only state translation and the migration infrastructure remained to be implemented, see Figure 2. State translation is the required marshalling and demarshalling of the schedd's captured data for execution at the target site. While the migration infrastructure is the combination of many components, including state translation, to logically checkpoint the schedd, move it to the target site, and restart it.

5.1 Install/Uninstall

A core set of functionality required by logical checkpointing is packaging all the service's components into a single file, cleaning up the packaged components, and unpacking this single file into a set of service components at the target site. This can be accomplished using an archiver, an installer, and an uninstaller

5.1.1 Archiver

A schedd service is more than just a binary, it is composed of several parts including the condor_config, jobQ, individual job files, and debugging logs. For ease and efficiency these files need to be packaged into a single file for compression and transmission. We have implemented the Condor Schedd ARchiver, *csar* to collect the needed components, package, and compress them into a single file. condor_config details the locations of all of the files needed by the schedd, including the location of supporting binaries. csar queries the condor_config to locate all of the needed files, copies them into a new directory, archives the directory, and compresses the archive. Archiving and compression are implemented using GNU's tar [7] and gzip [3] programs.

5.1.2 Installer

Upon arriving at a target machine the schedd must be installed. We implemented the Condor Schedd Installer, *csi*, to uncompress, unpack, and install a schedd from an archive. csi either installs the components in the current working directory, or, if supplied with an installation configuration file, installs the components in user-defined locations. Additionally, csi generates a log file detailing the location of all of the successfully installed components.

5.1.3 Uninstaller

A schedd may be migrated from one machine to the next. It would be poor design to assume that the schedd would eventually be migrated back to any machine it was migrated from. Once a schedd leaves a machine it must leave no files behind. Imagine a scenario in which several schedds made a brief stop at single machine; the machine's disk space would eventually fill with old schedd files. We implemented the Condor Schedd Uninstaller, *csu*, to clean up and remove schedd components. csu locates the components to uninstall by querying condor_config or an uninstall configuration file. In the event that an installation fails, csi can be used to clean up the partially installed schedd by supplying it with the log file created by the failed installation.

5.2 Marshall/Demarshall

In order for a service to migrate, its state must be made independent of its host environment. However, in order for the service to operate at the target host, its state must be attached to the new host environment.

5.2.1 JobQ Marshalling

The schedd exports its dynamic state in the form of a log file called the jobQ. The jobQ is a marked up version of ClassAds; the markings are used to indicate operation type: insert, delete, and update. The jobQ is dependent on the host machine of the schedd, specifically on the file system. Each file, including the binary, of each job is listed in the jobQ. At submission, a job's files are copied into a special schedd directory. The jobQ is updated to point to the special directory version of the files needed by a job. During migration the jobQ must be modified to remove these absolute file paths and thereby remove the dependency on the host file system. We refer to this modification as marshalling the jobQ. Every instance of an absolute file path that specifies a schedd component is replaced by a macro. For example the path to a mobile job's initial working directory may be represented as `Iwd "/scratch/condor/spool/cluster5.proc0.subproc0"`. The prefix `/scratch/condor/spool` specifies the spool component of the schedd and marshalling should replace the prefix with `$(SPOOL)`. The entry would then appear as `Iwd $(SPOOL)/cluster5.proc0.subproc0`. After arrival at the target machine the jobQ must be relinked to the file system so that the file paths point to an actual location. We refer to this relinking as demarshalling the jobQ. Every macro indicating a schedd component is replaced with the actual location of the component.

We implemented a program, *jobqMD*, to both marshall and demarshall a jobQ. The set of possible Condor ClassAds attributes may expand or old attributes may take on new meaning. To prevent this from requiring a rewrite of the jobQ marshaller a jobqMD configuration file stores which attributes are candidates for marshalling/demarshalling. To perform the marshalling and demarshalling correctly jobqMD requires a mapping from schedd component macros to their current location. This mapping can be directly computed from the installation log file both at the source and the target sites.

5.2.2 Configuration Marshalling

A schedd's static data is maintained in the condor_config file. This file specifies locations of schedd components, the name of the central manager, security settings and preferences, file system domain, and other settings. A subset of the properties defined in the condor_config file are machine independent. For example, the central manager will be the same machine regardless of where a schedd migrates within a pool. Similarly, the security settings should not change simply because the schedd has migrated. However, some properties are tied directly to the host machine. The schedd uses the properties defined in condor_config to locate its helper binaries, e.g. the shadow.

In order to migrate the schedd we must remove these host machine dependencies. Marshalling the condor_config file is different from marshalling the jobQ, in that, marshalling the jobQ introduced place holders for machine dependent attributes while marshalling condor_config simply removes any machine dependent properties. Demarshalling the condor_config reintroduces these machine dependent properties for the new host.

We implemented a program, *configMD*, to marshall and demarshall the condor_config. The set of properties stored in condor_config can change with each version of Condor, in that new features are added that require configuration variables and old features are removed. To handle this flexibility configMD uses a configuration file that details which properties are machine dependent. Additionally, each machine dependent property must specify whether it is a required or optional property. During demarshalling configMD must be provided with a file mapping required machine dependent properties to machine specific values. In the case where configMD finds a required machine dependent property with no entry in the map file during demarshalling, it produces an error and exits. If an optional machine dependent property is not in the map file, configMD does not include it in the demarshalled condor_config. configMD allows a series of macros to be used in the map file including macros representing the current machine's IP address and DNS name.

5.3 Schedd Checkpoint and Restart

Using the components from the previous section a schedd can be checkpointed and restarted on different hosts.

5.3.1 Schedd Checkpoint

A schedd must be shutdown prior to logical checkpointing, one cannot guarantee a correct logical checkpoint of the state if the state is changing. The jobQ is marshalled using the current installation log as a schedd component map file. Then the condor_config is marshalled with configMD. The schedd is then archived by csar and the schedd components removed by csu. What remains is a single logically checkpointed schedd. These tasks are completed by a program called *schedd_checkpoint*.

5.3.2 Schedd Restart

To restart a schedd, it must be installed at the target site from an archive using csi. The installation log is used as an input into jobqMD to demarshall the jobQ. The condor_config is demarshalled using configMD with either a predefined map file or a map file generated from the installation log and local knowledge about the target host. Finally, the schedd is restarted. The schedd reconfigures itself by reading the condor_config, then replays the jobQ to reconstruct its dynamic state, and, finally, registers its new location with the match maker. These tasks are completed by a program called *schedd_restart*.

5.4 Schedd Migration

Determining when to migrate is a service-dependent policy decision. We leave policy decisions about when to migrate to the user. Determining where to migrate is also a policy decision. If a user has a specific host in mind then they merely need to execute schedd_checkpoint on the source host, transfer the archive to the target host, and execute schedd_restart. However, the user may wish to migrate the schedd to a more general set of hosts with more complex requirements.

A user wishing to migrate the schedd only has to checkpoint the schedd, create a job submission file for the schedd_restart program with the checkpoint file as a parameter, and submit the job to another schedd. The checkpointed schedd will be restarted on the next available execute machine. Further, a user can include with the job submission a complex set of target machine requirements and preferences, detailing everything from average CPU load to a specific subset of target machines. Many of these steps are repetitive for every migration of a schedd. To ease the use of this type of migration we created a program,

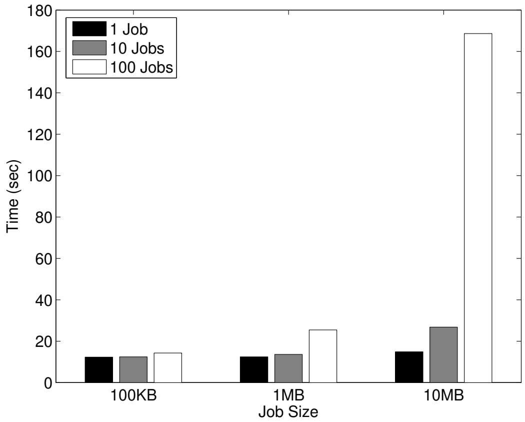

Figure 3: Checkpoint times for a schedd with varying number and size of jobs. *Note:* The x-axis is log-scale

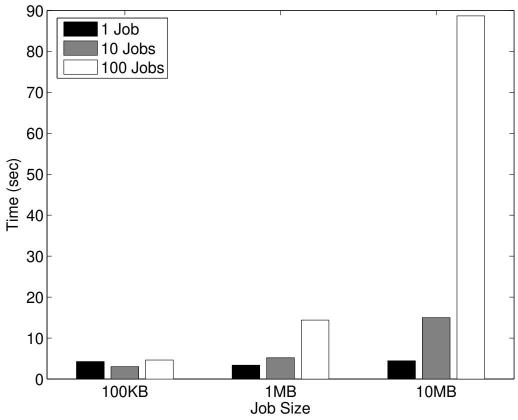

Figure 4: Restart times for a schedd with varying number and size of jobs. *Note:* The x-axis is log-scale

schedd_migrate which automates many of the steps. The user must only specify the schedd to migrate, the schedd to submit the job to, and the preferences and requirements for the migration. schedd_migrate executes schedd_checkpoint, generates the job submission file with the appropriate preferences and requirements, and submits the schedd_restart job. The astute reader may be wondering, how one can submit anything to a schedd if the schedd service on a machine has already been shutdown and checkpointed. Condor allows remote submission to a schedd. The schedd_restart job can easily be submitted to a schedd on another host.

6 Performance Analysis

To ensure that the cost of our checkpointing technique is not overly restrictive we performed a series of benchmarks. These benchmarks were run on a machine with a Intel P4 2.40GHz processor, 512MB of RAM, running the Linux 2.4.21 kernel.

Figure 3 illustrates the average checkpoint times of a schedd over five runs with 1, 10, and 100 jobs of size 100KB, 1MB, and 10MB. Job size, as referred to here, is the total size of a job's executable and required input files. Breaking these tests down into micro-benchmarks of the individual components we found that csar dominates the checkpoint time, accounting for over 90% in all cases. jobqMD and configMD together complete in less than 150 milliseconds even in the worst case.

Figure 4 displays the average restart times of a schedd over five runs with 1, 10, and 100 jobs of size 100KB, 1MB, and 10MB. Breaking these tests down into micro-benchmarks of the individual components we found that csi dominates the restart time, accounting for over 90% in all cases. jobQMD and configMD together complete in less than 60 milliseconds even in the worst case.

These figures show that logically checkpointing and restarting the schedd scales linearly with the size and number of jobs. Logically checkpointing a schedd with 100 10MB jobs, which yields a 1GB compressed schedd image, still completes in less than 180 seconds. A schedd can be restarted from this same image in less than 90 seconds. Given Condor's emphasis on high throughput computing we feel that these times are within reason. Our micro-benchmarks indicate areas of improvement, namely in csar and csi. A simple improvement to increase performance may be to not compressing the schedd image. We introduced compression to reduce migration overheads due to network transmission. However, when migrating within a fast local network it may be more efficient to send an uncompressed image.

7 Related Work

Service Continuations [28] and service migration in the Kea kernel [30] both provide kernel space for a service to store its state. After migration this state is returned to the service. Both of these implementations assume uniform access to the home file system. Service Continuations assumes a uniformly mounted distributed file system, while Kea only provides migration of a service within a single machine, between user and kernel space. Luo and Yang present the idea of zero-loss web services [17], however, their work is less service migration and more service fail-over as requests for a service are duplicated to a a backup service. Network address rewriting is used to switch from a primary service to the backup in during failure.

Process migration is closely related to service migration. Sprite [12] and Charlotte [10] are examples of in-kernel implementations of process migration. Sprite and Charlotte assume identical system images on both the source and target machines, including uniform file access. Additionally, Sprite has residual dependencies on the source site that allows users to query the source site about processes that have been migrated.

Zap [20] is a kernel module that allows groups of processes to be migrated without breaking their network connections. Zap even virtualizes the file system to allow a uniform view if the migration location can access the process's home file system. However, Zap does not include any of a process's files in the checkpoint so it cannot handle migrations outside the range of the process's home file system. Further, a failed Zap checkpointed process could not be safely restarted from the checkpoint because the files the process depends on may have been modified by the process after checkpoint but prior to failure. The Condor system also provides process checkpointing [16]. However, this checkpoint mechanism relies on remote I/O back to the source site to provide a uniform view of the file system.

Internet Suspend/Resume [15] uses VMWare [9] to checkpoint and migrate an entire operating system. This requires a process to run in a virtual machine even when it will never migrate. Further the current implementation of VMWare only supports x86 architectures.

The Tui System provides a migration architecture that dynamically captures and translates a processes data into machine independent form at run-time [26]. However, this approach relies on a specialized compiler and only works on type-safe programs.

8 Conclusion and Future Work

We have developed a set of techniques for checkpointing a service that removes the checkpointed data's dependencies on the host machine. We have created a multi-component architecture to apply this technique in mobilizing the Condor High Throughput System's distributed scheduler, the schedd. Further, we have used Condor to schedule the placement of the migrated service based on an arbitrary user-defined set of requirements and preferences. Additional work will be needed to determine whether these techniques are broadly applicable to other services. However, we feel that many services built using the crash-only approach proposed by Candea and Fox [11] will be able to take advantage of these migration techniques.

Our techniques for service migration do not provide a seamless migration in that there may be a brief interruption in service. In the future we may extend these techniques to include technologies such as MobileIP [21], SIP [31], or VNAT [27], to provide uninterrupted service migration. Our current implementation of the mobile schedd includes an operating system and machine architecture independent marshalling of the schedd's state. However, this state must be packaged with the schedd architecture specific binaries at checkpoint time which limits the architectures and operating systems the schedd can be restarted on. We intend to modify the schedd installer to ftp the appropriate binaries for the target site's operating system and architecture during installation. This way a schedd's restart will only be limited by the platforms supported by Condor and available at the ftp site.

Acknowledgments

Thanks to Alain Roy whose incite, comments, and paper reviews made a marked improvement in the quality of this publication. Thanks also to the rest of Condor Team whose patient support proved invaluable.

References

[1] *Apache Struts*, http://struts.apache.org/.

[2] *Emacs*, http://www.gnu.org/software/emacs/.

[3] *Gzip*, http://www.gnu.org/software/gzip/.

[4] *Java 2 Platform Enterprise Edition*, http://java.sun.com/j2ee/.

[5] *Kazaa*, http://www.kazaa.com/.

[6] *Microsoft .NET*, http://www.microsoft.com/net/.

[7] *Tar*, http://www.gnu.org/software/tar/.

[8] *The Apache Jakarta Tomcat*, http://jakarta.apache.org/tomcat/.

[9] *Vmware*, http://www.vmware.com.

[10] Yeshayahu Artsy and Raphael Finkel, *Designing a process migration facility: The charlotte experience*, IEEE Computer **22** (1989), no. 9, 47–56.

[11] George Candea and Armando Fox, *Crash-only software*, 9th Workshop on Hot Topics in Operating Systems, May 2003.

[12] Fred Douglis and John K. Ousterhout, *Transparent process migration: Design alternatives and the sprite implementation*, Software - Practice and Experience **21** (1991), no. 8, 757–785.

[13] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, *Exokernel: An operating system architecture for application-level resource management*, Symposium on Operating Systems Principles, 1995, pp. 251–266.

[14] Ian Foster and Carl Kesselman (eds.), *The grid: Blueprint for a new computing infrastructure*, Morgan Kaufmann, 2003.

[15] Michael Kozuch and M. Satyanarayanan, *Internet suspend/resume*, Fourth IEEE Workshop on Mobile Computing Systems and Applications, April 2002.

[16] Michael Litzkow and Marvin Solomon, *Supporting checkpointing and process migration outside the unix kernel*, Proceedings of the Winter 1992 USENIX Conference (San Francisco, CA), January 1992, pp. 283–290.

[17] Mon-Yen Luo and Chu-Sing Yang, *Constructing zero-loss web services*, 20th IEEE Intl. Conference on Computer Communications, June 2001.

[18] Sun Microsystems, *RPC: Remote Procedure Call Protocol specification: Version 2*, RFC 1057 (Informational), June 1988.

[19] Paul V. Mockapetris and Kevin J. Dunlap, *Development of the domain name system*, SIGCOMM, 1988, pp. 123–133.

[20] S. Osman, D. Subhraveti, G. Su, and J. Nieh, *The design and implementation of Zap: A system for migrating computing environments*, 5th USENIX Symposium on Operating Systems Design and Implementation, December 2002, pp. 361–376.

[21] C. Perkins, *IP Mobility Support for IPv4*, RFC 3220 (Proposed Standard), January 2002, Obsoleted by RFC 3344.

[22] Larry L. Peterson and Bruce S. Davie, *Computer networks*, second ed., Morgan Kaufmann, 2000.

[23] Raghu Ramakrishnan and Johannes Gehrke, *Database management systems*, third ed., McGraw-Hill Science/Engineering/Math, 2002.

[24] Rajesh Raman, Miron Livny, and Marvin Solomon, *Matchmaking: Distributed resource management for high throughput computing*, Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7) (Chicago, IL), July 1998.

[25] Mendel Rosenblum and John K. Ousterhout, *The design and implementation of a log-structured file system*, ACM Transactions on Computer Systems **10** (1992), no. 1, 26–52.

[26] Peter Smith and Norman C. Hutchinson, *Heterogenous process migration: The Tui system*, Software - Practice and Experience **28** (1998), no. 6, 611–639.

[27] Gong Su and Jason Nieh, *Mobile communication with virtual network address translation*, Tech. Report CUCS-003-02, Columbia University, February 2002.

[28] Florin Sultan, Aniruddha Bohra, and Liviu Iftode, *Service continuations: An operating system mechanism for dynamic migration of internet service sessions*, 22nd International Symposium on Reliable Distributed Systems, 2003.

[29] Douglas Thain, Todd Tannenbaum, and Miron Livny, *Distributed computing in practice: The Condor experience*, Concurrency and Computation: Practice and Experience (2004).

[30] A. Veitch and N. Hutchinson, *Dynamic service reconfiguration and migration in the Kea kernel*, CDS '98: Proceedings of the International Conference on Configurable Distributed Systems, IEEE Computer Society, 1998, p. 156.

[31] Elin Wedlund and Henning Schulzrinne, *Mobility support using SIP*, WOWMOM, 1999, pp. 76–82.