Towards a Service Migration Architecture for Service Availability

Yanjun Zuo
University of North Dakota, Grand Forks, ND, USA

Abstract - This paper presents our work-in-progress towards system architecture to support service migration in a devastating attack scenario. In our design, service migration is a security mechanism that transfers critical services from a compromised platform to other clean, healthy platforms. The architecture is to ensure that critical services will be continuously provided even the underlying platform has been damaged. Any system with the specified design can provide a level of guarantee that the critical services will be available in spite of malicious attacks and security incidents. We specify the components of such service migration-based system architecture and describe the functions of those components.

Keywords: Service migration, architecture, security, availability, critical service.

1 Introduction

Software intensive systems have been increasingly used in many high security and high integrity settings, including financial services, national defense, and healthcare, to cite a few. Due to their ultimate importance, critical systems are often the targets of malicious attacks. Even the best-designed systems cannot guarantee that well-organized attacks would never be successful. Given their crucial missions, it is important that those systems have the ability to adapt to the operating environment and continuously provide essential services to users even part of the systems has been compromised. From an engineering perspective, well-designed system architecture is the most fundamental factor in supporting a system’s security functions to resist malicious attacks and to mask system faults.

In this paper, we propose architecture to equip a system with the ability to migrate the critical services from a severely compromised platform to other clean, healthy platforms in a devastating attack scenario. The idea is that even the underlying platform has been damaged, the critical services can still be provided on other healthy platforms if an effective migration mechanism has been put in place. From a user’s perspective, service migration is transparent and the service is continuously available with little interruption. Conceptually, a service migration involves suspending the current service state on the old platform, moving the core service programs and trustworthy space to other platforms, and resuming where computation was left off on the new platforms. Migration has become the heart of the architecture of many safety-critical systems [1].

As compared with other survivability strategies, such as component/services redundancy and diversity [2-3], fault tolerance [4-5], and system reconfiguration [6-7], service migration is a viable solution to ensure that the most critical services are available in a security incident. Service migration is particularly effective in situations where damage to a system is so severe and very limited time is allowed to recover the damaged platforms without noticeably affecting service availability to users. In this research, we study the system architecture and essential components to support an effective service migration. We specify the main functions and properties of those components.

The rest of the paper is organized as follows. Section 2 discusses related work. The proposed system architecture and system components are presented in Section 3. We describe the data structures, functional specifications of the system components, and communications among them in detail. Section 4 concludes this paper.

2 Related work

System/service survivability has been an important topic in the research community. A set of techniques have been developed to improve a system’s ability to survive malicious attacks and/or system failures. Although architectures for system and service survivability ([8-13], to cite a few) have also been proposed, we have not found any specific research which explicitly describes system design and specification to support an efficient service migration. We attempt to bridge this gap in this research by proposing service-migration based system architecture to provide a reasonable level of guarantee that a system can fulfill its mission in a timely manner in the presence of attacks, failures, or accidents. Next, we discuss some related work on service/process migration.

Service/process migration has been proven to offer substantial benefits for fault resilience, dynamic load balancing, and service availability. In [14], service migration is proposed as an effective approach for service placement for ad hoc or autonomic networks in a dynamic environment. Instead of solving continuously a large optimization problem requiring global information, moving the service position, based on local information, towards more effective positions would be a more viable solution. The authors develop a service migration policy for undirected tree topologies. It is shown that the information available at the current service node is sufficient for determining the direction towards nodes with monotonically decreasing service provision costs. The proposed policy moves the service continuously towards the
optimal position in every step and reaches the optimal position through a shortest path migration trajectory. As the optimal position may change in a dynamic environment, the proposed policy adapts the service migration path continuously towards the currently optimal position.

In [15], a system for transparent process migration, called Zap, is proposed, which provides a thin virtualization layer on top of the operating system that introduces a Process Domain (pod) abstraction. The virtualization is integrated with a checkpoint-restart mechanism that enables processes within a pod to be migrated as a unit to another machine. A prototype is implemented in Linux that supports migration of unmodified applications without any kernel modification. In [16], a transparent mechanism is presented for commodity operating systems that can checkpoint multiple processes in a consistent state so that they can be restarted at a later time. An algorithm has been developed to record process relationships and save and restore shared states in a manner that leverages existing operating system kernel functionality. In [17], a reliable, flexible and efficient application checkpoint-restart is presented, which aims for inclusion in mainline Linux kernel. The approach is implemented in the operating system kernel and can detect resource leakage outside containers.

Different from the above research, our focus in this research is service-migration based system architecture. Our work complements the existing research by proposing system design for resisting malicious attacks and masking security damage. The main design objective of the proposed architecture is to make sure that the critical services are continuously provided even in a challenging environment.

3 System architecture to support service migration

The system we consider in this paper is a distributed system with an overall architecture as shown in Figure 1. The two control components are Proxy Server and Service Migration Manager. Each of them has a set of sub-components with specialized functions. The services provided by the system are hosted on a set of platforms. In the following sections, we first discuss the system specification in terms of those platforms and services offered by the system. Then, we discuss the functions of the two control components in facilitating an effective service migration.

3.1 Platforms and services

There are a set of platforms \( P_1, P_2, \ldots, P_n \), distributed throughout the system to provide services to users. Each platform \( P_i \) represents a virtualized computing environment. Technically, it can be a single server machine, a cluster of computers (e.g., a server farm), or a computing infrastructure in a cloud computing structure (the notation of Infrastructure as Service). \( P_i \) has both volatile and stable storage and can support a set of services as represented by its capability list, denoted as \( \text{Abi}(P_i) \).

The system offers a different types of services \( S_1, S_2, \ldots, S_m \). Each \( S_j \) is provided through a set of service programs. Consider a cloud infrastructure operated by a financial institution such as a bank. The system offer multiple types of services such as online banking, stock trading, investment and mortgage management, and customer relationship management. For each type of service, there may be a set of service instances corresponding to different users (internal or external). We use \( s_{j,1}, s_{j,2}, \ldots, s_{j,m} \) to represent the \( m \) service instances of a service type \( S_j \). For example, for an online banking service type, there may be several users who are conducting online banking activities. While all those users’ operations are supported by the same set of service programs, each user application has its private data space. Each user application’s data, the corresponding private memory space, and the underlying shared service processes form a service instance. To receive a service, a user must be authorized. After a successful authorization, an instance of the requested service (type) is created for the user. Each service instance is self-contained – the service processes and the user data can be referenced within the namespace assigned to the service instance. As we will see later, this type of encapsulation makes service migration easier from one platform to another. Some virtualization approaches are readily available such as Zap [15], OpenVZ [18], and Xen [19]. As a functional requirement, each service instance \( s_{j,i} \) is executed only on one platform at any given time. Furthermore, each service type \( S_j \) is assigned a priority level \( \text{Priority}(S_j) \) based on the functional specification under which the service is to be executed and on the significance of the service type. The priority of a service instance inherits the priority of its parent service type.

3.2 System control components

3.2.1 Proxy server

The Proxy Server provides an interface to users. Its behavior is represented as a finite state machine as shown in Figure 2. The Proxy Server contains two operating units and one data table (called Service Instance Registration Table (SIRT)). The first unit, the Authorization Manager is responsible for authorizing service requests based on user identities, service certificates, or other service credentials. Since authorization is not unique to service migration, our design does not specify any particular authentication approach that a system must use. Rather, any proven access control mechanism can be applied depending on the application under consideration. Once a user request for a service type \( S_j \) has been authorized, another unit, the Service Operation Manager checks the System Resource Database (SRD) to verify that there is a platform in the system that can provide the requested service. As shown in Figure 3, SRD maintains one entry about each platform \( P_n \), including such information as the capacity list of \( P_n \), i.e., \( \text{Abi}(P_n) \) (the types of services that \( P_n \) can support) and resources available on \( P_n \) to provide services to users. Each platform \( P_i \) has limited resources in terms of memory storage, CPU cycles, and network connections and it can only support a limited number of service instances.
Service quality is also a factor to consider when determining the platform load. For example, a platform in the financial institution’s system may only be able to support up to 1,000 concurrent online trading service instances while maintaining a satisfied level of service quality. Any additional request for this type of service will go beyond the capacity of the platform and thus must be re-assigned to a different platform. If a platform $P_i$ has been identified satisfying both the functional and resource requirements, the requested service will be assigned to be provided by $P_i$. Operationally, the Service Operation Manager sends a “Service Instance Deployment” message to $P_i$, instructing it to deploy and execute the requested service.

Upon receiving the service deployment message, $P_i$ creates a new service instance $s_{i,k}$ (here we assume that the new service stance is the $k^{th}$ instance of service type $S_i$ on the platform $P_i$). To customize services provided to different users, each service instance may be executed differently. $P_i$ generates the corresponding execution plan and sends the plan to the Proxy server to be recorded in SIRT. An execution plan represents a concise description of how each service instance is executed. In its simple form, the execution plan includes the

| Platform $P_i$ | Service Capability List $A_i(P_i)$ | Resources Available |

---

![Figure 1: The Service Migration System Architecture](image1.png)

![Figure 2: State Machine Representation of the Proxy Server](image2.png)

![Figure 3: Scheme of System Resource Database (SRD)](image3.png)
versions of the service programs and the algorithms (if any) to execute the service instance, the software signature of those programs, and other important parameters and variables to execute the service instance. As we will see, the execution plan provides important information about the execution details of each service instance. This information is very useful in re-generating the service programs in order to continuously execute \( s_{j,k} \) on a new platform should the original service programs have been severely damaged (more will be discussed in Section 3.2.2). The Proxy Server creates a new entry for \( s_{j,k} \) in SIRT to bind the service instance to the assigned platform. An entry in SIRT for \( s_{j,k} \) is shown in Figure 4. Basically, the entry keeps track of the service instance (i.e., \( s_{j,k} \)), its service type (i.e., \( S_j \)), the platform on which it is executed (i.e., \( P_i \)), the execution plan of the service instance, and an initial empty execution service status. Afterwards, the status of the service instance, including important intermediate steps of the service execution, is periodically updated by \( P_i \).

![Service Instance Registration Table (SIRT)](image)

### 3.2.2 Service migration manager

The Service Migration Manager supervises the entire service migration process. Its behavior is represented as a finite state machine as shown in Figure 5. It contains four operating units: Damage Assessor, Migration Scheduler, Resource Manager, and Migration Gateway. The functions of those components are described next.

a) **Damage assessor**

A service migration is triggered by an event such as the detection of damage on a platform \( P_i \). The Damage Assessor analyzes the collected incident data and decides whether the damage on \( P_i \) is severe enough so that a service migration is the best action to take given the current situation. If the faults are not severe and thus can be tolerated or masked (through mechanisms such as system/service redundancy, self-healing, or reconfiguration), the system’s operations can be continued with a minimum level of delay. However, if the number of fault types is large or different actions/events must be taken following error detection, fault tolerance and damage masking become difficult. In the latter case, service migration is a viable solution for minimizing the impact of malicious attacks in order to continuously provide crucial services.

The damage assessment results suggest two types of migration for a service instance: (1) a **heavyweight migration**, or (2) a **lightweight migration**. A heavyweight migration should be applied if the underlying platform has been severely damaged but the main functions of the service programs are relatively tamper-free. Therefore, the service programs are still trustworthy. In this case, the entire service programs, along with the private spaces of the individual service instances, are migrated in one contained package. However, if the number of the service instances is too large for the destination new platform to handle (the capacity of a platform can be determined based on SRD), then multiple new platforms are required to host those service instances. In this situation, the individual service instance data (including their private namespaces and operating status) are migrated to different new platforms along with a copy of the service programs to each platform. On each of those new platforms, the service programs can continuously support the migrated sub-set of service instances. Different from a heavyweight migration, a lightweight migration is applied if the service programs on the compromised platform have also been damaged. Therefore, the entire service space is no longer trustworthy. In this situation, it is necessary to generate new service programs on the those migrated platforms. Those newly generated service programs must be able to provide the functions offered by the original ones. Technically, new service program generation requires the execution plan and the status data of each service instance. Once the new service programs are established, the service instances can be continuously executed. Service program generation and service set up on the new platforms will be discussed in more detail in Section c).

b) **Migration scheduler**

Once a service migration is decided to be the best action to take, the Service Scheduler executes a function \( \text{Choose}(P_i, s_{j,k}, P_{k}) \) to generate a feasible arrangement for each critical service instance \( s_{j,k} \) with a high priority (e.g., \( \text{Priority}(s_{j,k}) \geq L^* \), where \( L^* \) represents a threshold for high service priority) to be migrated from its current platform \( P_i \) to a new, healthy platform, say \( P_k \). In the meantime, the service programs are halted on \( P_i \), which consists of freezing service processes, recording global data (service configuration and state), recording the states of individual processes, and terminating the entire service programs. Due to limited resources and time constraints, other less critical services on the compromised platform \( P_i \) may be temporarily disabled and resumed after the damage is contained – a strategy to ensure that the most critical services will be guaranteed in a challenging environment.

A migration process is composed of three sub-actions: (1) migration preparation, e.g., saving the service programs and private data for each service instance in a resumeable image in a self-contained format with header, global data, internal process dependencies, and shared resources (e.g., task structure, open files); (2) service/data transfer, e.g., synchronizing data copies, withdrawing transactions, establishing recovery points, and disseminating the packed service programs and service instance data to the new platforms; and (3) service setup on the new platforms, e.g., resuming the service spaces and configuring the necessary service instance parameters. When a service instance is started on the new platform, a new namespace has to be created and its configuration/state will be restored. For the underlying
service programs, the inter-dependency relationships of individual service processes have to be re-established. To provide an ensured service migration, the system must preserve the invariants of process inter-dependencies on the new platform, maintain the consistency of memory contents, and ensure the integrity of transactions. The priority, response time, and throughput of each critical service instance must be ensured (or appropriately adjusted) during and after a service migration.

c) Resource manager

The Resource Manager manages migration related resources (e.g., the platforms that are capable of providing a particular type of service, the current capacities of those platforms), which are crucial for a timely scheduling plan for an effective service migration. When one service instance is migrated to a new platform, the Resource Manager must notify the Proxy Server of this re-binding between the migrated service instance and its new platform. Consequently, the Proxy Server updates the corresponding entry in SIRT. Other important functions of the Resource Manager include setting up the migrated services on the new platforms. More specifically, there are two main functions: (1) service program generation (in case of a lightweight migration); and (2) fuzzy data generation and recovery in case that some service data has been damaged. They are discussed in more detail next.

Service program generation. As we mentioned earlier, a lightweight migration moves the execution plan and status data of a migrated service instance to a new platform without moving the service programs (which have been damaged). To continuously execute the service instances, new service programs must be generated and deployed on the new platform. In our design, service program generation is conducted by a software factory. As shown in Figure 6, the factory works like a software manufacturer: taking raw materials (supplied by the Resource Manager) – service functional specifications, execution plan of the service instances, and service quality and security requirements, and then assembling and producing a set of service programs to support the migrated service instances. The new service programs are then uploaded to the new platform, and they must provide the essential functions of those original service programs running on the old platform. The new software programs should have different implementations by applying such techniques as randomized code, obfuscation, and diversity. Therefore, the new software programs are not subject to the same type of attacks as the old ones. Afterwards, the execution of the service instances can continue from where they have been left.

Fuzzy data generation and recovery. Since certain data of a migrated service instance $s_{ij}$ may have been damaged, that data cannot be used on the new platform. Conducting a full cycle, offline data recovery may be difficult given limited...
time to make the service available. To deal with this situation, the Resource Manager must provide supplemental data by generating fuzzy data to support the continuous functions of $s_j$. Fuzzy data means that the data may not be accurate but are safe to use for a temporary time period to ensure the critical services are continuously available. In other words, fuzzy data is only meant to be used temporarily when the service instance is being executed on the new platform - as the critical functions must remain operational at all times. At a later time when the environment improves, an accurate recovery must be performed as soon as time and resources permit so that the fuzzy data items reflect exact values, not just acceptable values. Consequently, all transactions that have used approximate values have to take necessary corrective measures to ensure data integrity. Those operations are supervised by the Resource Manager since data is the most important resource for many services. Previous research [20-21] has demonstrated that fuzzy data can be generated in real time with a minimum level of delay.

d) Migration gateway

Service migration must be fast and secure. In our design, dedicated network communication channels have been set up among the platforms as preserved pathways for the purpose of service migration. Those channels are either established dynamically or pre-configured between two platforms. Since those channels are used only for transferring service packages from one platform to another and operating at a highly controlled model, they are relatively secure and efficient. The Migration Gateway is responsible for identifying and setting up those migration channels before or during a service migration. If necessary, this unit may act as an intermediate gateway to forward the migration data to the destination platform. There have been a set of techniques [22-23] available for configuring reliable connections among two nodes in a network.

4 Conclusions

In an attack scenario, service migration is a viable solution for minimizing the impact of attack effects in order to continuously provide critical services to users. In this paper, we presented system architecture to support an efficient service migration for the system to provide a reasonable level of guarantee that the services will be continuously available in a challenging environment. We have specified the system components and described their properties and functions in supporting an assured service migration. Our work lays out a foundation for further study in building survivable and reliable systems in high security and high integrity settings.

Acknowledgment

This material is based upon work supported by the US Air Force Office of Scientific Research (AFOSR) under Award FA9550-12-1-0131. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of AFOSR.

References


