Static vs. Dynamic Recovery Models for Survivable Distributed Systems

Joon Park
School of Information Studies
Syracuse University, NY, USA
jspark@syr.edu

Pratheep Chandramohan
Dept. of Computer & Information Sciences
Syracuse University, NY, USA
pchandra@syr.edu

Abstract

The need for survivability is more pressing for mission-critical systems, especially when they are integrated with other COTS products or services. As information systems became more complex and the interdependence of these systems became higher, the survivability picture became more complicated. In this paper, we introduce two basic models with respect to static and dynamic models, where each model has restart and continue modes. We describe each approach and compare the trade-offs in terms of simplicity, resource efficiency, adaptation, service downtime, immunization, and robustness. Later, we introduce a hybrid model by combining these two basic models. Our approaches are transparent to the client side and can be applied to existing distributed systems by extending the server side with the components necessary for the schemes.

1. Introduction

Complex interconnected information systems and network infrastructures are increasingly vulnerable to loss of services. In some cases, lives and livelihoods depend on the continuation of IT systems that monitor and maintain critical activities such as traffic control, telecommunication, banking and power management. As information systems become more complex and we rely more on their correct operation, the need for survivable systems becomes ever more pressing. Furthermore, the interdependence of these systems complicates the survivability picture. For example, even some sensitive military systems depend on public and private communication networks for both function and administration, thus a network failure could cause them to become unusable and unmanageable [PBBSM02]. Indirectly related subsystems may be integrated to support a high-level mission, but problems with such a subsystem could affect the entire mission.

The definitions of survivability have been introduced by previous researchers [KS00, LF99]. In this paper, we define survivability as the capability of an entity to continue its mission even in the presence of damage to the entity. An entity ranges from a single component (object), with its mission in a distributed computing environment, to an information system that consists of many components to support the overall mission. An entity may support multiple missions. Damage can be caused by internal or external factors such as attacks, failures, or accidents.

It is not always possible to anticipate every type of failure and cyber attack in large information systems, and attempting to predict and protect against every conceivable failure and attack soon becomes too cumbersome and too costly. Additionally, some damage result from novel, well-orchestrated, malicious attacks that are simply beyond the abilities of most system developers to predict. Under these conditions, even correctly implemented systems do not ensure that the system is survivable.

Although advanced technologies and system architectures improve the capability of today's systems, we cannot completely avoid threats to the systems. This becomes more serious when the systems are integrated with Commercial Off-the-Shelf (COTS) products and services, which usually have both known and unknown flaws that may cause unexpected problems and that can be used by attackers to disrupt mission-critical services. Usually, organizations (including the Department of Defense (DOD)) use COTS systems and services to provide office productivity, Internet services, and database services, and they tailor these systems and services to satisfy their specific requirements. Using COTS systems and services as much as possible is a cost-effective strategy, but the tailored systems also inherit flaws and weaknesses from the COTS products and services that they use. Traditional schemes for ensuring survivability with respect to a set of anticipated failures and cyber attacks do not meet the challenges of

0-7695-2056-1/04 $17.00 (C) 2004 IEEE

providing assured survivability in these systems that must rely on commercial services and products.

In this paper, we identify static and dynamic survivability models where each model has two different modes (restart and continue), and we discuss their trade-offs. The dynamic model recovers the infected components (objects) with immunized components, which are not vulnerable to the same type of failure or attack in the future. The comparison of the trade-offs between the models leads to the hybrid survivability model, which provides more robust survivability services than other two models.

The rest of this paper is organized as follows. In Section 2, we summarize a survivability strategy from our previous work. In Section 3, we describe the static and dynamic survivability models. In Section 4, we discuss the trade-offs of those survivability models, and in Section 5, we introduce a hybrid model that combines the static and dynamic models for providing more robust services. Section 6 concludes the paper.

2. Survivability Strategy

The damage caused by cyber attacks, system failures, or accidents, and whether a system can recover from this damage, will determine the survivability characteristics of a system. A survivability strategy can be set up in three steps: protection, detection and response, and recovery [PF02].

First, we must attempt to protect the target system against possible damage by using conventional and novel security technology, environmental measures, and operational protection measures [KW96, RVH81]. A survivable system should resist cyber attacks (e.g., denial of service), internal failures (e.g., human errors, software design errors, hardware malfunctions, etc.), and external accidents (e.g., natural disasters). Cryptographic technologies, access control mechanisms, and security devices can be used for this purpose. Ensuring that security patches have been installed and that approved configurations of COTS products and services are used protects the system from known attacks and failures. This will also entail obtaining various authorizations for operation. For instance, denying the attacker intelligence about the system configuration and its assets protects the target system against a well-planned and executed cyber attack. We believe that a good survivability solution covers not only what happens when an anticipated damage occurs but also what might happen in response to unanticipated events.

Knowing that COTS products and services have unmitigated and unknown vulnerabilities, we must anticipate that our protection measures will sometimes be ineffective. For instance, we cannot guarantee that a program from a known source (e.g., checked by the digital signature) is free of malicious code. Furthermore, we cannot guarantee that systems are configured correctly. The second leg of the survivability strategy is to detect and respond to damages. Every layer of the computational infrastructure must be monitored, and detection responses will be correlated to obtain a better understanding of the threat environment. Log analyses programs, intrusion detection systems, and anti-virus programs can be used for this purpose. When damage is identified, a proper response must be mounted to protect system resources, such as isolating the host under attack, partitioning the system, redirecting the attacker to a honey pot, or simply alerting security managers of the impending threat. Providing the capability to strengthen protection mechanisms in response to observed threat behavior could prove to be quite effective. However, protection and response measures may prove insufficient to allow the system to continue operations.

Therefore, the third leg of our strategy is to be able to recover some degree of capability and tolerate some classes of damages. When the damage has been adequately thwarted, it is necessary to assess the damage and to recover mission-critical capabilities and services [LAJ00, JMA99, KED98, PWS01, WBS00]. To continue the mission, we may not need to recover all the damaged components completely in the original shape. Rather, we need to recover the capabilities and services that affect the mission in a degraded mode. Some damage may not be detected immediately, so the damages may propagate until they are detected. Recovery depends on the severity of the damage (i.e., how many resources have been affected, etc.) and the recovery strategies and remaining undamaged resources that are in place. Identification of critical system capabilities, comprehensive damage assessment, recovery schemes for different degrees of damage severity, and redundant resources for reconfiguration can ensure the ability to continue operation despite partial damages. The survivability models described in this paper are mainly based on static and dynamic recovery schemes.

To make a system survivable, it is the mission of the system, rather than the components of the system, to survive. This implies that the designer or assessor should define a set of critical services of the system to fulfill the mission. In other words, they must understand what services should be survivable for the mission and what functions of which components in the system

should continue to support the system's mission [MEL00]. Specific survivability requirements typically vary from one application to another, hence it is important to understand a comprehensive set of survivability requirements for the mission before survivability management. A system designer or assessor may need a top-down view of the critical functions of the components in the system in order to make decisions about the survivability strategies and their impact on the overall system.

3. Survivability Models

In this section we discuss different survivability models. We categorize the models into three types namely static, dynamic, and hybrid models. We describe the static and dynamic models in detail in this section and later develop a hybrid variety (described in Section 5) combining the aspects of both static and dynamic models. Detailed comparisons between the static and dynamic models are provided in Section 4. Although our approaches can be applied to any critical components in a distributed computing environment, for simplicity we describe our approach with a server component in this paper.

3.1. Static Survivability Model

The recovery scheme of this model is based on redundant servers, prepared before the operation, to support critical services continuously to the client in a distributed client-server environment. The redundant servers can be located in the same machine or different machines in the same domain or even different domains. Existing approaches such as dynamic reconfiguration can be associated with this static model. Although the term "dynamic" is used in the terminology, it belongs to the static model according to our definition, as long as the available components are generated before the operation starts. The same service of a server can be provided by identical servers (e.g., copies of the original servers) or different components that are implemented in different ways. Isolated redundancy (in different machines or domains) usually provides higher survivability, because the replaced server can be running in an uninfected area. For instance, if the redundant servers are distributed in different network places, the services provided by the servers can be recovered (in the event of network failures) in different environments. However, if there is a failure within a server, replacing the server with an identical copy is not effective, because identical components in the same environment are vulnerable to the same internal failure.

Figure 1 shows the basic scheme of the static model in a UML (Unified Modeling Language, [BJR98]) collaboration diagram, consisting of five major components: client, server, registry, broker, and monitor. Normal operation steps are depicted by the messages from 1.1 to 2.6 in Figure 1. The client component may have a regular user interface embedded in the client application. The server side needs to be extended with additional communication channels among the components for supporting survivable services. The registry provides a conventional naming service in a distributed computing environment. The redundant server components are generated and deployed before the operation starts. The servers register themselves to the registry implying that their service is available to client applications. The monitor maintains a list of available servers, checks the servers' operating status, and responds to the brokers' request for server status information. Furthermore, the monitors communicate with other monitors to check the status of the realms in the other monitors' coverage. A realm can vary from a set of component through machines to whole domains. For instance, if Machine X becomes unavailable because of internal failures or cyber attacks, this status are detected by X's monitor and informed to other monitors, including Machine Y's monitor. Therefore, when it is necessary to replace Server Z in Machine Y because of its abnormality, the Server Z's monitor in Machine Y does not consider Machine X as a safe place.

The broker acts as an entry point to the service. The client interface will first connect to the broker (instead of registry first) asking for a specific service, thus making the recovery process transparent to the client. The broker keeps a list of active servers running, through the monitor. The list is updated periodically with the monitors' reports. The broker chooses a server for the client request from the active list of running servers based on the service context, such as workload of the available servers or network traffic, to maximize the level of the service. Since all the connections to the server go through the broker, the broker can act as a load balancer. By load balancing we can distribute the connections (load) to the maximum available servers so that individual loads to each server are minimized. Then the broker queries the registry for a particular server stub, which is then returned to the client immediately. A server stub is a server proxy, which marshals and un-marshals the client requests to the server and back. The client can communicate directly to the server through the server stub without intervention of the broker from this point.

After the direct link between the client and server has been established, they communicate in a traditional distributed way. Before initiating the transaction the client should authenticate with the server. For more trusted communications, mutual authentication between the client and server should be provided. After a successful authentication, the client requests transactions and the server responds with the appropriate results based on the client's privileges. The server decides whether the client is authorized to access its functionality. This mechanism can also be extended among other components in the model. Authentication and authorization services are required, but are not described in detail here, because we focus on the survivability schemes in this paper.

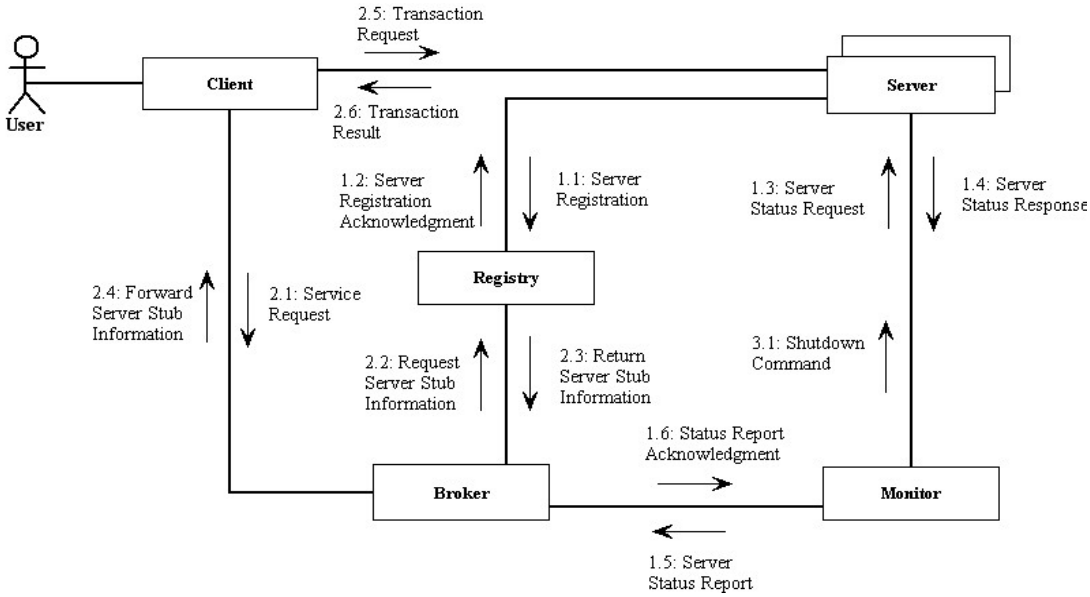

Figure 1. Static Survivability Model

If all the mission-critical objects are running in their normal states, the operation ends with message 2.6 in Figure 1. However, in the event of any failures (component, system, or network), security breaches, or attacks on the system, the recovery operation starts with message 3.1 as depicted in the figure. The components (we use a server in our description for simplicity) can be totally or partially infected. The server's erratic behavior will be discovered by the monitor, which has been checking the server's status. If the server is in an abnormal state (e.g., if the server response is very slow or shows unexpected results) then the status is detected by the monitor. A simple way to check the server's status is to ping the server periodically (or when it is necessary) and analyze the response results. However, the ping mechanism does not provide detailed information about the abnormality and the results are between the server and the monitor, not between the server and the client. More powerful monitoring mechanisms support more accurate decisions about the abnormality. When recovery is necessary, the monitor sends a shutdown message to the infected server. In the meantime, the broker chooses an available server for the client' request from the active server list based on the service context, and helps the client to be reconnected to the new server. From this point, we consider two different modes: *restart* and *continue*. The continue mode is capable of continuing the client's transaction transparently by restoring the client's transaction state. The restart mode does not restore the client's transaction state, thus the client needs to restart the operation from the beginning of the transaction.

3.1.1. Restart Mode. The client's transaction state information is not stored in this mode. When the connection between the client and server breaks in the event of failures or attacks, there is no way to recover to the old state by the new server after the connection is restored.

An implementation of the restart mode is typically simpler than that of the continue mode because we do not need to provide state-restoring mechanisms. Furthermore, it can be more secure or stable than the

continue mode in some applications because it does not store (reuse) any session information, which may be sensitive and possibly compromised by attacks or may have caused failures. The components do not share the transaction state information with each other. All of the transaction before the point of breaking the connection is lost. The transaction has to be restarted from the beginning when the connection with the replaced server is achieved.

From the user's point of view, this is less convenient than the continue mode because the user starts the transaction from the beginning and repeats the procedures he followed before. Therefore, if there are frequent component reconnections involved in the same session, the restart mode is not reasonable. This mode can cause confusion among the users and may cause a critical mistake in the system because of the possibly repeated operations.

For instance, let us take an electronic banking system example where a user performs a deposit transaction. If the connection is broken in the middle of the transaction right after the user deposits but before the session is over, then the user needs to repeat the transactions (here it is depositing) with the newly assigned server. Similarly, when the client's request is delayed (e.g., due to network congestion) reaching the server, the client may conclude that the server did not respond to its request and the client might generate the same request again. However, as far as the server is concerned, the request is made twice. Typically, most systems keep and update the clients' transaction states information after each session. This cannot solve the problem completely. To avoid the server processing the client's request multiple times without maintaining the client's transaction state, we employ sequence numbers in each client request. Thus if two consecutive sequence numbers are out of order or the same, then the system can sense that the operation is inappropriate and react based on pre-defined policies.

3.1.2. Continue Mode. In the continue mode, the server stores the client's transaction state information. So when a transaction is recovered, it is enough for the replaced server to start from the point where the transaction is broken. This is possible because the transaction state information is stored and shared among the related components in the system. In this mode, the monitor stores fine-grained transaction information of each transaction session. When the client is connected to a new server, the broker will be able to provide the transaction state information to the new server through the monitor. Therefore, the new server can restore the transaction to the point where it was broken and continue it. From the user's point of view, this is more convenient than the restart mode because the user doesn't need to repeat the transaction procedures from the beginning when the server is reconnected. Let us take the same example we discussed in the restart mode. A user makes a deposit transaction to his account. If the connection breaks (due to network problem or server attack), what happened in the restart mode is that the user had to restart the whole transaction. In the continue mode, however, the user need not restart the whole transaction. Instead, the user may not even notice that the server was reconnected, which provides transparency to the user.

The transaction state data should be stored in persistent storage (at least until the transaction is finished) and will be shared later during recovery. In our implementation, state information is stored in the form of records. Each record has fields for transaction ID, timestamp, sequence number, and a snapshot of the internal variable's state. A sample of an internal variable would be the balance amount in the case of an electronic banking example. We store the state information in XML files at the monitor side. Alternatively, the information can also be logged into a relational database (e.g., Oracle, MS-SQL etc.) for a large-scaled system. The advantage of using a relational database over XML is that it allows dealing with large volumes of data without compromising speed and robustness.

3.2. Dynamic Recovery Model

In this model, the components, which caused failures, with failures, or under attack, are replaced by dynamically generated components on the fly and deployed (in runtime) by the factory as and when they are required. The server components can be generated and deployed in any machine inside the distributed network, which may span multiple domains. Changing the runtime environment of the same object will be helpful for survivability to some level, as we described previously. Furthermore, this model can replace the infected components with immunized components, which provides more robust services than the static model. Detailed comparisons between the static and dynamic models are provided in Section 4.

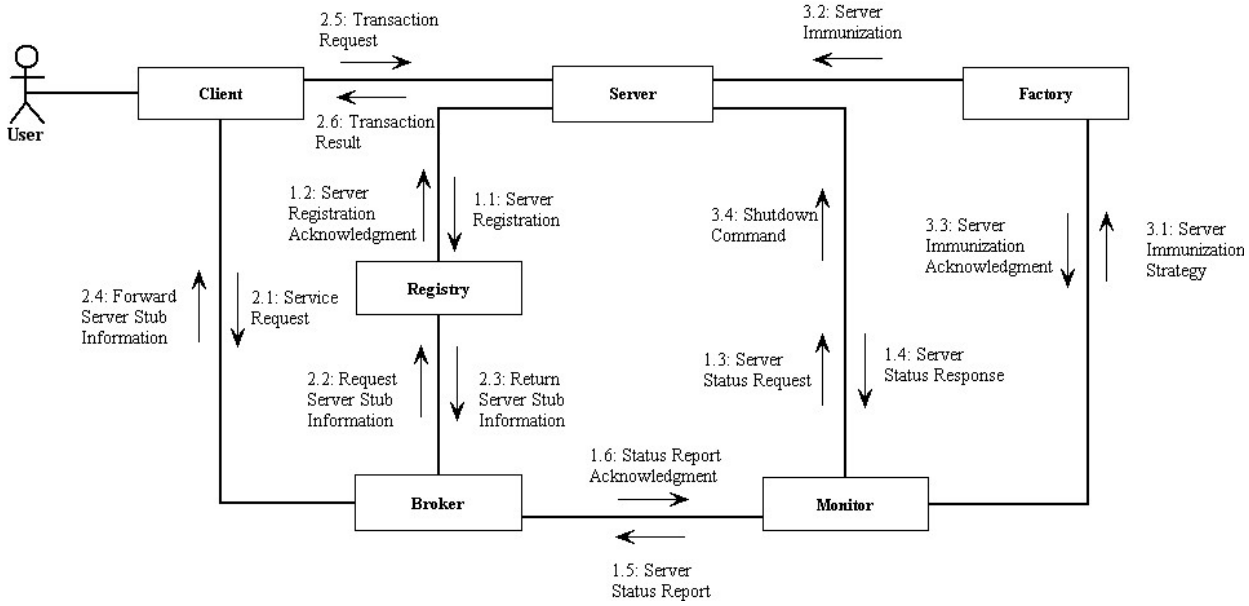

Figure 2. Dynamic Survivability Model

Figure 2 shows the basic scheme of the dynamic survivability model in a UML (Unified Modeling Language, [BJR98]) collaboration diagram. It consists of six major components: client, server, registry, broker, monitor, and factory. Note that there is one additional component, factory, to the five components used in the static model. The concept of the factory was originally introduced as one of the commonly used design patterns in the object oriented design [GHJV94]. In this model, the operational scenario in a normal state and the functions of the components are similar to those in the static model. However, the recovery scheme is different (depicted by the messages from 3.1 to 3.4 in Figure 2).

Unlike the static model, there is no redundant component in this scheme and the factory generates new immunized components that will be replaced by the infected components. The monitor performs operations such as gathering the server status report, forwarding it to the broker and, most important, analyzing the reason for failure and the type of attacks in order to find out possible immunization methods. If the monitor knows the reason for the abnormality, immunization can be more accurate. For instance, if the server's monitor figures out that the problem is caused by loading a malicious client object into the infected server during the runtime, the server's monitor commands the components in its realm not to accept the client's requests in the future and informs other monitors of this. It is the responsibility of the malicious client's monitor either to fix the problem in the client or to make it inactive. When the monitor detects there is an intrusion to a server in its realm, it asks the corresponding factory to generate an immunized server and deploys it into a safe area (from which correct monitoring responses are coming), while the monitor issues a command to shut down the old server. In order to connect to the new server, the client needs the stub of the new server. This can hide the location of the new server from the attacker using different binding information. Although the attacker figures out the new location of the server, it cannot send any messages to the server, because the communication port number of the server and even the interfaces have been changed.

If the monitor does not know the exact reason for the abnormality or if it is hard to immunize components against the known failures or attacks, the factory simply replaces the infected component with a new one in a safe environment. We call this a generic immunization strategy. Although this approach supports the service availability to the client continuously, the new server might still be susceptible to the same failures or attacks. In the case of failures, we can avoid possible common problems such as resource sharing conflict if the factory deploys the new server in an idle machine. The resource-sharing conflict is not easily detected or predicted in the programming stage, but it frequently occurs in runtime in distributed computing environments. Generic immunization is more effective against cyber attacks. If a component (a machine or a whole domain) is under attack, the generic immunization strategy asks the factory to generates a new copy of the component and deploys it in a new environment, which is safe from the attack.

From this point, as we did in the static model, we can consider two modes: *restart* and *continue*. In restart mode, the monitor does not maintain transaction state information. In continue mode, the monitor maintains transaction state information, which is passed to a newly built component, so that the transaction is recovered to the point where it failed. The characteristics of the two modes are discussed in Sections 3.1.1 and 3.1.2.

4. The Trade-Offs

In this section we discuss the trade-offs of the static and dynamic survivability models. Table 1 shows a summary of the characteristics of each model.

Table 1. A Comparison of Survivability Models

	STATIC SURVIVABILITY MODEL	DYNAMIC SURVIVABILITY MODEL	HYBRID SURVIVABILITY MODEL
Simplicity	Higher	Medium	Lower
Resource Efficiency	Lower	Higher	Medium
Adaptation	Pre-fixed	Dynamic	Pre-fixed + Dynamic
Service Downtime	Shorter	Longer	Shorter
Immunization	Environments	Environments + Components	Environments + Components
Robustness	Lower	Medium	Higher

The static survivability model is the simplest solution. This model is architecturally similar to existing fault-tolerant approaches, using redundant resources. There are *n* numbers of redundant servers running in parallel that are deployed even before the operation starts. If one server fails or falls prey to intruders, then the broker delegates incoming requests to another server in the remaining server pool. Since the redundant components are ready to be used (unlike in the dynamic model) during the failure or attack, the service downtime is relatively shorter than the one in the dynamic model.

Basically, the strength of the static model depends on the prepared redundancy, which brings up the question of "how many" redundant components we need to provide. Technically, we could maintain as many redundant components as necessary for a critical service. However, if the initially selected component is running in its normal stage, which means there is no need to use other redundant components, the cost for running the redundant components has been wasted. We say in this situation that the resource efficiency is low, which also means that the maintenance cost is high. The static model provides limited adaptation capability against failures and attacks based on the reconfiguration among prefixed alternatives. For instance, we can configure the redundant components to be running in different environments (e.g., different machines or domains). These redundant components basically provide the same service, but their implementations can be different, so the adaptation can be provided by choosing a different component running in a different environment for the same service among the prefixed configurations. If there is one component available for a specific situation in the remaining server pool, adaptation is possible, but if there is not, it is impossible. The component immunization is limited by means of new environments, while the component itself cannot be immunized on demand as the dynamic model provides. Therefore, overall robustness of the static model is lesser than that of the dynamic model.

On the other hand, the dynamic survivability model is more robust, although its implementation is more complex because of the need for a powerful immunizing facility such as monitors or factories. This model is based on dynamic component building. The service downtime of the component is longer than that of the static model because of the overhead for building components. The backup component is not readily available as it is in the static model. We conducted a series of experiments to calculate the actual downtime in our implementation. The experiments were run in the restart mode on a machine with the following configuration: Intel Pentium 2GHz, 512 MB RAM, and Windows XP. The average size of the server components is approximately 250 lines of code in Java and RMI (Remote Method Invocation). For the static model the average downtime was 32ms while for the dynamic model it was 2187ms. The downtime of the static and dynamic model includes the time taken by the monitor to switch the path from the client to the new

server. However, the downtime of the dynamic model also includes the time taken for the factory to construct and deploy the server components, to build new stub classes, and the time taken for the client to download the new server stub (if new stub is different from old stub), as well as the switching time. In the dynamic model, resource efficiency is high because the resources (components) are generated only when necessary, which requires a lower maintenance cost than that of the static model. The model can adapt dynamically to the kind of failures or attacks that occur and can create a new component for a particular situation, based on the monitor's analysis at that moment. This provides resistance to the same kinds of future failures and attacks. Immunization can be performed by reinforcing the components as well as deploying them in new environments. Therefore, the overall robustness is higher than that of the dynamic model.

5. The Hybrid Model

We discussed the trade-offs of the static and dynamic models in Section 4. The Hybrid survivability model combines the features of those two models. The dynamic model has an inherent disadvantage in terms of service downtime. The recovery process can range from seconds to few minutes. This downtime drawback will cause major problems in mission-critical systems because there will be no service available for clients during the recovery period. On the other hand, the static model has inherent disadvantages in terms of resource efficiency, adaptation, and robustness.

To compensate for the weaknesses in the two models, we incorporated the idea of a hybrid model. In the beginning of the operation, an array of *n* redundant components is initiated as described in the static model approach. These redundant servers will be used as a buffer while a more robust server is generated and deployed by the dynamic model approach. When a server fails because of an attack or internal failure, the buffer servers will take over the service for a brief period until the new immunized server is initialized. Since all of the buffer servers are susceptible to the same failures or attacks, it is a matter of time before the redundant servers are also infected by the same reason. Therefore, if the transition period is long, multiple buffer servers may be used before the immunized server is ready. The monitor informs the factory about the server failure and requests to build a new immunized server. The factory, upon receiving the request from the monitor, starts to build the immunized server. During the building time the factory deploys a buffer server to serve clients for a temporary period. The factory updates the active server list to the monitor, and the monitor then updates the broker concerning the server list. In the meantime, the broker delegates all service calls to the temporary buffer server until it is notified about the construction of an immunized server. After building the immunized server, the factory deploys the new server, and the buffer server is revoked from service. Again, the monitor is informed about the creation of the immunized server, which in turn informs the broker about the deployment of a new server. To make the service more robust, the factory can generate multiple copies of the immunized server for possible use in the future. The hybrid model ensures the availability of service to the client and provides more robust services in the end. However, this model needs more complex implementations than the two other models previously discussed.

6. Conclusion and Future Work

In this paper we have identified static and dynamic models where each model has restart and continue modes to support the pressing requirements for survivability in mission-critical systems, especially which are integrated with COTS products and services. We then made a comparison between the static and dynamic models, discussing the benefits and limitations of each in terms of simplicity, resource efficiency, adaptation, service downtime, immunization, and robustness. The comparison led to the hybrid survivability model, which provides more robust survivability services than the other two models. Our approaches are transparent to the client side and can be applied to existing distributed systems by extending the server side with the components necessary for the schemes.

7. Acknowledgment

Our thanks to the support from the New York State Center for Advanced Technology in Computer Applications and Software Engineering (CASE).

8. References

[BJR98] Booch, G., Jacobson, I., and Rumbaugh, J. 1998. The unified modeling language user guide. Addison-Wesley.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of reusable Object-Oriented Software*, Addison Wesley, 1994.

[JMA99] S. Jajodia, C. McCollum, and P. Ammann. *Trusted Recovery*. Communications of the ACM, 42(7), pp. 71-75, July 1999.

[KED98] J. Knight, M. Elder, and X. Du. *Error Recovery in Critical Infrastructure Systems.* Proceedings of the 1998 Computer Security, Dependability, and Assurance (CSDA'98) Workshop, Williamsburg, VA, November 1998.

[KS00] J. Knight and K. Sullivan. *Towards a Definition of Survivability*. Proceedings of the 3rd Information Survivability Workshop (ISW), Boston, MA, October 2000.

[KW96] W. Wulf, C. Wang, and D. Kienzle. *A New Model of Security for Distributed Systems*. Proceedings of the New Paradigms in Security Workshop, Lake Arrowhead, California, 1996.

[LAJ00] P. Liu, P. Ammann, and S. Jajodia. *Rewring Histories: Recovering from Malicious Transactions.* Distributed and Parallel Databases, 8(1), pp. 7-40, January 2000.

[LF99] H. Lipson and D. Fisher, *Survivability -- A New Technical and Business Perspective on Security.* Proceedings of the New Security Paradigms Workshop (NSPW'99), Caledon Hills, Ontario, Canada, September 21-24, 1999.

[MEL00] N. Mead, R. Ellison, R. Linger, et al. *Survivability Network Analysis Method*, SEI Technical Report: CMU/SEI-00-TR-013, September 2000.

[PBBSM02] Walter Perry, Robert W. Button, Jerome Bracken, Thomas Sullivan, Jonathan Mitchell. *Measures of Effectiveness for the Information-Age Navy: The Effects of Network-Centric Operations on Combat Outcomes.* ISBN: 0-8330-3139-2, MR-1449-NAVY.

[PF02] Joon S. Park and Judith N. Froscher. *A Strategy for Information Survivability*. 4th IEEE/CMU/SEI Information Survivability Workshop (ISW), Vancouver, Canada, March 18-20, 2002.

[PWS01] P. Pal, F. Webber, R. Schantz, M. Atighetchi, and J. Loyall. *Defense-Enabling Using Advanced Middleware- An Example*. Proceedings of MILCOM 2001, Maclean, VA, October 2001.

[RVH81] N. Roberts, W. Vesely, and D. Haasl. *Fault Tree Handbook*. NUREG-0492, Office of Nuclear Regulatory Research, 1981.

[WBS00] J. Wylie, M. Bigrigg, J. Strunk, G. Ganger, H. Kiliccote, and P. Khosla. *Survivable Information Storage Systems*. IEEE Computer, August 2000.