Incorporating Fault Tolerance Tactics in Software Architecture Patterns

Neil B. Harrison
University of Groningen, Utah Valley University
800 West University Parkway
Orem, Utah 84058 USA
+1 801 863-7312
neil.harrison@uvu.edu

Paris Avgeriou
University of Groningen
PO Box 407
9700 AK Groningen, The Netherlands
+31 50 3237057
paris@cs.rug.nl

ABSTRACT

One important way that an architecture impacts fault tolerance is by making it easy or hard to implement measures that improve fault tolerance. Many such measures are described as fault tolerance tactics. We studied how various fault tolerance tactics can be implemented in the best-known architecture patterns. This shows that certain patterns are better suited to implementing fault tolerance tactics than others, and that certain alternate tactics are better matches than others for a given pattern. System architects can use this data to help select architecture patterns and tactics for reliable systems.

Categories and Subject Descriptors

D.2 [**Software Engineering**]:; D.2.11 [**Software Architectures**]: Patterns; D.4.5 [**Reliability**]: Fault-tolerance

General Terms

Reliability

Keywords

Patterns, Software Architectures, Fault-tolerance, Reliability tactics

1. INTRODUCTION

One of the chief challenges in designing reliable systems is that the overall structure and behavior of the system – its architecture – is tightly linked to its fault tolerance. Decisions made about the architecture of the system impact the ease with which the system can be made reliable. Conversely, decisions about how to implement fault tolerance features in the system can impact, and even shape the architecture.

It is clear that fault tolerance must be a key consideration during the early phases of software development – early attention to fault tolerance contributes to a system that supports fault tolerance. In order to understand this better, let us distinguish between two different development scenarios, green field development and legacy system enhancement.

In green field projects, the architecture of the system is not yet fixed. One can create a software architecture that supports fault tolerance, as well as the functional requirements of the system and the other important quality attributes. In fact, we found in a study that when architects design a new system, they consider both functional requirements and quality attribute requirements (such as fault tolerance) together, and make architectural design decision based on both kinds of requirements [15]. In green field projects, the fault tolerance requirements can and should help shape the structure of the system's architecture.

On the other hand, most software projects are enhancements to legacy systems. In these projects, the architecture of the system already exists, and is usually very difficult to substantially change. However, measures to improve fault tolerance are still needed; measures must be enhanced, or new measures added. For example, one of the authors added measures to improve fault tolerance in a large, mature telecommunications system [16].

While it is necessary to improve the fault tolerance of legacy systems, it may not be easy. The existing structure of the system cannot easily be changed; instead fault tolerance measures must be implemented within the framework of the existing architecture. Depending on the architecture, this can be easy or difficult. This study focuses mainly on architecture patterns already in a system, so it applies mainly, but not exclusively, to existing systems.

An important way that an architecture affects the fault tolerance of a system is by making easy or hard to implement measures that improve fault tolerance. However, for any given measure to improve fault tolerance, we do not know which architectural structures make it easy or difficult to implement that measure. Furthermore, we do not understand why a fault tolerance measure is easy or difficult to implement in a given architectural structure. This makes it difficult to make informed choices about alternative measures, and to assess the costs and benefits of fault tolerance measures. In green field projects, it makes it difficult to select the best architectural structures that support the desired fault tolerance measures. In short, our ability to effectively incorporate fault tolerance measures is compromised.

In order to gain this understanding, we studied fault tolerance measures and well-known architectural structures. For each architectural structure (architecture pattern), we examined several fault tolerance measures (called tactics.) We investigated how one

SERENE 2008, November 17-19, 2008, Newcastle, UK.

would implement the each fault tolerance tactic in each pattern. This study included what parts of the structure of the pattern would change in order to implement the tactic, and how they would change.

From this study we learned several things that are potentially useful for architects and developers. We found that certain architecture patterns were naturally a better fit for fault tolerance than others. We found that some of the tactics themselves are generally easier to implement in the well-known patterns than others; this can be especially useful when considering alternatives and cost-benefit studies.

In this study, we focus on fault tolerance as described by Avizienis et al.[2], where fault tolerance consists of error detection (identifying the presence of an error), and recovery (transforming a system state that contains one or more errors and faults into a state without detected errors and without faults that can be activated again.)

Part 2 describes software architecture and fault tolerance, and discusses in general terms the nature of their interaction. Part 3 describes background work. Parts 4 and 5 describe our study and its results. Part 6 discusses the use of the data, and Part 7 presents the data itself. Parts 8 and 9 discuss related and future work.

2. ARCHITECTURE AND FAULT TOLERANCE

It is well known that the architecture of a system impacts its fault tolerance. One important way that an architecture impacts fault tolerance is by making it easy or hard to implement measures that improve fault tolerance.

2.1 Fault Tolerance Tactics

Let us examine the measures taken to improve fault tolerance in more detail. Bass et al [3] define measures to improve quality attributes as tactics. There are two different types of tactics, designated as design time and runtime tactics. We describe design time and runtime tactics for fault tolerance below.

Design time tactics are measures that are applied across all parts of the system at design and coding time to enhance fault tolerance. They often take the form of design or coding rules, such as “check all return codes,” or “prevent buffer overruns.” Each developer must apply these tactics when designing and writing code.

On the other hand, runtime tactics are specific actions the system will take to achieve fault tolerance while the system is running. In particular, the system takes certain actions to detect faults and errors in the running system, prevent faults from impacting the integrity of the system, and recovering gracefully from faults if they do occur. Typical examples of fault tolerance runtime tactics include voting and transaction rollback.

In this study, we are interested exclusively in the runtime tactics for fault tolerance, because we studied the architectural impact of the fault tolerance tactics. The design time tactics have no large-scale architectural impact. Throughout the rest of the paper, references to “tactics” or “fault tolerance tactics” mean runtime tactics for fault tolerance.

Because these tactics are specific actions, they are implemented much like features: each tactic has a design, and is generally decomposed into components, connectors between the components, and required behavior. Thus it follows that the structure and behavior of a tactic impacts the structure and the behavior of the system. This is an important point at which fault tolerance (implemented via tactics) and the architecture meet.

2.2 Architecture Patterns

Architecture patterns are common architectural structures, which are well understood and documented [1][5][23]. These patterns describe the high level structure and behavior of general systems. Architecture patterns contain the major components and connections of the system to be built.

During architectural design, an architect may select one or more architecture patterns to follow to produce a system structure. The architect selects patterns based on their ability to support the requirements of the system, including fault tolerance requirements.

Patterns, then, embody the high level structure of the system. The structure of tactics is more local and low level. Therefore, the structure of the tactic must fit into the larger structure of the pattern.

2.3 Implementing Fault Tolerance Tactics in Patterns

Because of the constraints of architecture, we must consider implementing fault tolerance tactics in the context of the patterns used. Therefore, we must understand the nature of the implementing tactics in the architecture patterns. We need to understand the following:

- Given a certain pattern, what are the best fault tolerance tactics to use, based on ease of implementation?
- Why is one tactic easier to implement in a pattern than another tactic?
- How does one implement a tactic in a given pattern; what parts of the pattern must be modified?

In order to help us answer these questions, we studied numerous patterns and fault tolerance tactics.

3. BACKGROUND WORK

Because it has long been understood that the architecture of a system has an impact on its fault tolerance, the interaction of architecture and fault tolerance has been an important topic of study. Numerous architectural and process approaches for different aspects of fault tolerance, such as handling exceptions, have been proposed [9][7][10][18][8], or for fault handling [20]. General approaches to architecture and development of fault tolerant systems have also been proposed [4]. A comprehensive list of works that address architecting fault tolerant systems can be found in [21].

Numerous architecture patterns have been identified [1][5]. Some of the pattern documentation includes descriptions of the pattern’s impact on fault tolerance, although it is rather sparse and superficial. A very high-level summary of patterns’ impact on quality attributes including fault tolerance is contained in [13].

Patterns are modified by the implementation of fault tolerance features. Laibinis and Troubitsyna [19] discuss fault tolerance in a layered architecture. De Lemos, Guerra and Rubira [6] show an architecture pattern, namely C2, that is changed to handle exceptions. This is a detailed example of the types of changes described in this work.

Avizienis et al [2] note that dependability is achieved through fault prevention, fault tolerance, fault removal, and fault forecasting. Muccini et al [21] note that fault tolerance involves error handling, and fault handling. Our study covers common techniques of both.

There has been considerable interest in techniques of error handling and fault handling. Techniques in addition to exception handling have been identified, and have been occasionally labeled as patterns at a lower level than architecture patterns [11]. They are analogous to Bass et al's tactics. Specific fault tolerance actions have been defined for telecommunications systems [24], as well as for limited-memory systems [22].

In a previous study, we examined 47 system architectures to identify the patterns found in them [12]. As part of it, we identified several different general types of systems in this set, and identified the patterns most commonly found in each type of system. Three of these types are systems that often have high fault tolerance needs: embedded systems, dataflow and production systems, and information and enterprise systems. This gives an idea of several of the common architecture patterns in reliable systems. These patterns are Shared Repository, Layers, Pipes and Filters, Presentation Abstraction Control, Model View Controller, Broker, Client-Server, and State Transition. These patterns are fully described in [1][5][23].

4. THE STUDY

We began the study by identifying the patterns and tactics to examine.

We selected the patterns from Buschmann et al. [5], because they are among the best known architecture patterns – the ones people are most likely to use. We then added other patterns from our study of architectures, as described above. This added the Client-Server, Shared Repository, and State Transition patterns, giving eleven patterns in all.

For tactics, we used the tactics as given in Bass et al [3]. These tactics are a limited set, but are well known techniques for improving fault tolerance. In addition, they are well defined as tactics. A short description of each tactic appears in section 7. We studied all the tactics given in this book (13). These cover many general approaches to implementing fault tolerance.

We studied each fault tolerance tactic to determine how it is typically implemented in each architecture pattern. We did this by examining the structure and behavior of both the pattern and the tactic, and determining where the tactic would fit. We examined the question, if your system is using this particular architecture pattern, how would you implement this particular tactic? What components of the pattern would change, and how would they change? In this way, we attempted to characterize the nature of the interaction between the tactic and the pattern. We do note that our characterizations are based on analysis, heuristics and experience, which is of necessity somewhat subjective. In particular, difficulty of implementing a given tactic in a given situation is partly determined by the individual designer's experience and expertise.

Ultimately, the individual data is the most useful to developers; the complete data is contained in an appendix. However, we also examined the data as whole, looking for trends and generalizations.

4.1 Impact on Pattern Participants

Buschmann et al note that the structure of patterns consists of components and connectors, and call them collectively, participants. We note the following general types of impact on both components and connectors.

For each tactic and pattern pair, we identified which components in the pattern must be modified in order to implement the tactic, and how they must be modified. This, then, becomes a guide for implementing fault tolerance tactics: if you are using a particular pattern, this information helps you understand where your architecture must change, and what you have to do to it.

We found that a tactic impacts the individual components of a pattern, and impacts them in different ways. In short, to implement a tactic, one changes the components of the pattern. We found several types of changes, and noted that they have different impact on the components.

The following table shows the ways in which a tactic may impact a component of a pattern. These are arranged in order of increasing impact, i.e., the first one ("Implemented in") is the easiest to implement.

Table 1. Types of changes to pattern components

Type of Change	Description	Impact
Implemented in	Part of the tactic is implemented within a component, with no external change to the component. (A special case of Modify)	Only the behavior of the component changes. Generally the easiest to implement.
Replicates	A component is duplicated, with little or no change to its behavior. Usually done for redundancy. (A specialization of Add.)	Usually easy to implement.
Add, in the Pattern	A new component is added within the structure of the pattern (e.g., a layer is added in the Layers pattern.)	Generally easy or moderately easy to implement.
Add, out of the Pattern	A new component is added that is not part of the pattern structure, causing the system to deviate from the original pattern (e.g., adding a monitor to Pipes and Filters.)	Usually difficult to implement. Makes the pattern difficult to find, making maintenance more difficult.
Modify	The behavior and the	Impact varies: some

	structure of the component changes.	changes are trivial, but others are very difficult.
Delete	A component is deleted.	Never observed it. Impact would be large.

The changes to components and the changes to the connectors of a pattern are related. We note that the type of change to a connector is dictated by the type of change to the component, and is quite similar. We describe the changes to connectors in terms of the changes to the components, as summarized above.

Table 2. Types of changes to pattern connectors, based on changes to components

Type of Change to a Component	Corresponding Change to Connectors
Implemented in	No change
Replicates	Connectors added between replicated components and other components. These connectors may be within the structure of the pattern
Add, in the pattern	New connectors added, within the pattern structure
Add, out of the pattern	New connectors added, outside the pattern structure
Modify	New or modified connectors needed. Probably outside the pattern structure
Delete	Would have to remove connectors

We focused our attention on the components, for two reasons. First, as visible above, the connector changes are similar to the component changes. Second, the sources of the patterns focus mainly on connectors. We recognize that connectors may need consideration outside of components, though, and note it in future work.

4.2 Impact on Patterns

The impact of implementing a tactic on a pattern as a whole is the aggregate of the impact of the tactic on the pattern's components. Of course, a tactic itself consists of components, so one must consider what components a tactic contains, and how they might be implemented in the pattern. The difficulty of implementing the tactic's components becomes the impact on the pattern.

We defined a five-point scale to describe how difficult it is to implement a particular tactic in a given pattern. It is based on the impact on the components of the pattern. The descriptions follow:

1. Good Fit (+ +): The structure of the pattern is highly compatible with the structural needs of the tactic. Most or all of the changes required are the "Implemented in" type, and the behavior of the pattern and tactic are compatible. Any structure changes ("Modify") required are very minor. For example, the Broker architecture strongly supports the Ping-Echo tactic because the broker component already communicates with other components, and is a natural controller for the ping messages.
2. Minor Changes (+): The tactic can be implemented with few changes to the structure of the pattern, but the changes are minor and more importantly, are consistent with the pattern. These types of changes are "Replicates" or "Add, in Pattern." Structure changes ("Modify") are minor. For example, the Layers pattern supports the active redundancy tactic by replicating the layers, and adding a small distribution layer on top. Although another layer is added, it is entirely consistent with the pattern.
3. Neutral (~): The pattern and the tactic are basically orthogonal. The tactic is implemented independently of the pattern, and receives neither help nor hindrance from it.
4. Significant Changes (-): The changes needed are more significant. They may consist of "Implemented in," "Replicates," and "Add in Pattern" where behavior changes are substantial. More often, they include significant "Modify" or minor "Add out of Pattern" changes. For example, the Presentation Abstraction Control manages simultaneous user sessions. Implementing a Rollback tactic would likely require significant extra code to ensure that different interfaces are synchronized.
5. Poor Fit (- -): Significant changes are required to the pattern in order to implement the tactic. These consist of significant "Modify" and/or "Add out of Pattern" changes. The structure of the pattern begins to be obscured. For example, introducing Ping-Echo into a Pipes and Filters pattern requires a new central controlling component, along with the capability in each filter component to respond to the ping in a timely manner.

It is important to note that the architecture of every system is different, and has different requirements and constraints, resulting in a unique structure. Therefore, the impact of a tactic on an architecture could be different from our findings. As such, these findings should be taken as general guidelines. They are most useful as guides for comparisons of tactics and of patterns, and as helps for implementation.

5. FINDINGS

We looked for general trends of both patterns and tactics.

5.1 Implementing Tactics

We noted general noted general cases of the changes to patterns caused by implanting tactics, as well as some special cases worthy of note.

We noted two special cases of changes. In the first, adding a component changed the pattern to a different pattern. In particular, most of the structural changes to the Client-Server pattern involve adding a controlling component, thus changing the structure into the Broker pattern. This is a straightforward change, because the two patterns are very closely related, and we marked it as a positive impact.

Second, the implementation of a tactic within a pattern resulted in the addition of a pattern to the architecture. The most striking case of this was the Layers pattern. The addition of any of the recovery-preparation and repair tactics (voting, active redundancy, passive redundancy, or spare) were applied, an additional controlling component was placed in front of duplicate copies of the layered system. This could create a combination of the Layers pattern and the Broker pattern, with the layered system behind the Broker component. (We note that in our studies of architectures [12], the Layers and Broker patterns were frequently found together.) In nearly all the cases where an additional pattern was added in conjunction with the implementation of a tactic, it was the Broker pattern that was added. We also called this a positive impact.

The fault tolerance tactics we studied were in four groups, as described in [3]:

1. Fault Detection: Measures to detect faults, including incorrect actions and failure of components.
2. Recovery – Preparation and Repair: preparing for recovery actions; in particular redundancy strategies so that processing can continue in the face of a component failure.
3. Recovery – Reintroduction: Tactics for repair of a component that has failed and is to be reintroduced.
4. Prevention: Preventing faults from having wide impact by removing components from service or bundling actions together so they can be easily undone.

Within each group, the tactics can be considered alternatives to each other (although some tactics, notably Exceptions and Transactions, may be applied in addition to others in the same group.) Each of the alternatives within a group has a similar objective, with a different design for achieving it. Because the designs are different, they may have differing impacts on the patterns. One tactic might have a larger or smaller impact on the structure of a pattern. This information can be used to help decide which tactic to use. The following table summarizes the difficulty of implementing tactics; namely the count of the implementation difficulty levels of the tactic in each pattern:

Table 3. Tactic Implementation Difficulty Summary

Group, Tactic	+ +	+	~	-	- -
Fault Detection					
Ping/Echo	3	2	5	0	1
Heartbeat	2	3	5	0	1
Exceptions	3	0	7	0	1
Recovery: Preparation					
Voting	2	5	4	0	0
Active Redundancy	4	5	1	1	0
Passive Redundancy	2	5	2	2	0
Spare	1	2	5	2	1
Recovery: Reintroduction					
Shadow	1	5	2	3	0
Resynchronization	3	3	2	2	1
Rollback	5	1	2	1	2
Prevention					
Removal from Service	1	1	7	2	0
Transactions	5	2	3	0	1
Process Monitor	1	2	6	2	0

We observed the following from the data: The fault detection methods of ping/echo and heartbeat are very similar. In most cases exceptions are basically orthogonal to the architecture. Within Recovery: Preparation, we see that active redundancy is the easiest to implement. Sparing is the most difficult. Within Recovery: Reintroduction, resynchronization and rollback are both strong, although there are a few patterns where rollback is difficult to implement. The easiest Prevention tactic is transactions. This gives an idea of which tactics are easiest to implement in general. However, it should be stressed that tactics within a tactic group are not completely interchangeable: other factors (such as other patterns or requirements) will influence the selection of tactics as well.

5.2 Suitability of Patterns for Fault Tolerance

The previous section describes how individual tactics impact the patterns. We saw that for some patterns, the impacts of the fault tolerance tactics were mostly positive, while for some other patterns, the impacts were mostly negative. This would indicate that some architecture patterns are better choices for reliable systems than others, all other factors being equal. The following table shows for each pattern, how many tactics have the different levels of implementation difficulty. They are arranged in approximate order of ease of implementation, starting with the pattern that is generally the easiest to add fault tolerance tactics to.

Table 4. Tactic Implementation Difficulty in Patterns, Summary

Pattern	+ +	+	~	-	- -
Broker	10	2	1	0	0
State Transition	8	2	3	0	0
Layers	3	7	3	0	0
Client-Server	2	8	3	0	0
Shared Repository	6	0	6	1	0
Microkernel	4	2	7	0	0
Model View Controller	0	2	9	2	0
Presentation Abstraction Control	0	2	9	2	0
Blackboard	0	3	6	2	2
Reflection	0	1	8	3	1
Pipes and Filters	2	2	1	3	5

We note the following from the data: In the Layers pattern, several tactics can be implemented by adding a new layer, which is consistent with the pattern. The Pipes and Filters pattern is somewhat bi-modal: tactics for fault detection are a poor fit, but tactics involving redundancy fit well. The Broker is very strong because many tactics involve a monitor component, which is a natural fit for the broker component. Client-Server is a good fit because it changes easily into the Broker pattern. Blackboard is hurt when it comes to fault tolerance because the AI nature of its processing makes it difficult to define states or points for synchronization or rollback. Reflection is generally not a particularly good fit for fault tolerance because of its emphasis on runtime configurability. Microkernel, Model View Controller, and Presentation Abstraction Control are all largely independent of fault tolerance actions. Shared Repository assumes the use of commercial databases, which implement many fault tolerance tactics already. So it is very strong. State Transition is a good match because several tactics rely on state behavior.

It is useful to consider how a group of tactics impacts a pattern. This gives more detail about what parts of fault tolerance are compatible or incompatible with various patterns. In the following table, we show the average difficulty for each group of tactics.

Table 5. Tactic Group Implementation Difficulty in Patterns, Average Score

Pattern	**Fault Detection**	**Recovery : Preparation**	**Recovery : Reintroduction**	**Prevention**
Broker	+	+ +	+ +	+ +
State Transition	++	+	+ +	+
Layers	+	+	+	+
Client-Server	+	+	+	+
Shared Repository	+	+	+ +	+
Microkernel	+ +	+	~	~
Model View Controller	~	+	~	~
Presentation Abstraction Control	~	+	-	~
Blackboard	~	~	-	-
Reflection	~	-	-	~
Pipes and Filters	- -	+	-	-

Because most tactics in a tactic group can be used alternatively, it is useful to look at the best fitting tactic for a pattern in each tactic group. In this table there is a more positive story: for most patterns there is some tactic in each tactic group that can be readily implemented.

This information can be used to help when deciding which patterns to use. However, if the architecture is already established, it can simply give a rough idea of the comparative amount of work needed to add fault tolerance actions and to maintain the system in the future.

Table 6. Tactic Group Implementation Difficulty in Patterns, Best Case

Pattern	**Fault Detection**	**Recovery : Preparation**	**Recovery : Reintroduction**	**Prevention**
Broker	+ +	+ +	+ +	+ +
State Transition	++	+ +	+ +	+ +
Layers	+ +	+	+ +	+ +
Client-Server	+	+	+ +	+ +
Shared Repository	+ +	+ +	+ +	+ +
Microkernel	+ +	+ +	~	+
Model View Controller	~	+	+	~
Presentation Abstraction Control	~	+	+	+
Blackboard	+	+	+	-
Reflection	~	-	~	+
Pipes and Filters	- -	+ +	+	~

5.3 Summary of Findings

Architectures generally follow well established architecture patterns. We found that some of the patterns are well suited for implementation of these fault tolerance tactics; in other words, some patterns are good fits for reliable systems. However, nearly all the popular patterns accommodate all these fault tolerance tactics, with a greater or lesser degree of modification to the pattern needed.

We found that one can identify the components in the architecture where the tactics can be implemented, and how much change is required. One can use this information to learn about the tactic's implementation, to select among alternate tactics, and even sometimes select different or additional architecture patterns. It also appears that this information is potentially important in the understanding of the interaction of the implementations of multiple tactics.

However, all potential uses depend on the availability of this information to the architects. While architects are generally experts (or at least should be), and should have much of this information in their heads, it is unlikely that have it all. We propose that it be collected and become part of a handbook for architects of reliable systems.

6. USING THE DATA

The data has several practical uses, particularly in the areas of architectural synthesis and evaluation, as described by Hofmeister et al [17]. We discuss each below. Of course, any architectural activity must consider not only how fault tolerance tactics affect other quality attributes and how other tactics or architectural solutions affect fault tolerance, but all quality attributes important to the system as well.

6.1 Architectural Synthesis

Architectural synthesis is the process of designing the architecture. Although it is usually considered in the context of designing a new system, it can also be considered for enhancing an existing system.

Let us begin with the case where a new system is truly a green field architecture. The architecture is still fluid, and the architect is weighing different options. In this case, the architect can examine different candidate architecture patterns to see which is the best fit for implementing the fault tolerance tactics that are required. This is an instance of Pattern-Driven Architectural Partitioning (PDAP), a process for leveraging architecture patterns and their interaction with the quality attribute requirements of the system [14].

The more common case is that the system is based on previous work, either a previous release of the same product, or similar projects. In this case, the architecture patterns are mostly fixed. In these systems, one can use the data in these ways:

- The data gives guidance for implementing a tactic in a given pattern. It gives pointers to which components might be changed, and how.
- It provides information about how readily different alternate tactics can be implemented in a pattern. This can help the developer make an informed decision about which tactic is a better fit for the patterns in the system.
- It can help the developer gauge how much work will be required to implement a particular tactic.
- It can help the developer understand the potential interactions of multiple tactics. Tactics which impact the same component of a pattern are likely to interact with each other, and deserve special attention.

6.2 Architectural Evaluation

This data can be used in architectural reviews to highlight areas of potential difficulty: areas where the patterns and the tactics do not match well.

A less obvious aspect of architectural evaluation is understanding the system. It has been estimated that up to 80% of a cost of a system over its lifetime is in maintenance, and up to 50% of the maintenance effort goes to understanding the system. We established in earlier work that one can identify architecture patterns in legacy system documentation [12]. However, patterns are changed by the application of tactics, thus making it more difficult to identify the pattern and understand why it was changed. As reviewers and maintainers understand the tactics used, they can more readily identify the modified patterns in the system, and gain insight into their implementations.

7. PATTERN AND TACTIC DATA

For the architect and developer, the key information is the data itself. In this section we describe the fault tolerance tactics we studied. We follow it with the data for the Layers pattern. The data for the rest of the patterns is available in an appendix.

7.1 Descriptions of the Tactics

There is not a universally accepted terminology for the various tactics of fault tolerance. Therefore, we give brief descriptions of the tactics and how they are implemented, as described in [3].

1. Tactics for Fault Detection:

   a. Ping/Echo: A monitoring component issues a ping message to one or more components under scrutiny, and expects to receive an echo message back within a predetermined time. If a component does not respond within the time limit, the monitoring component considers that component to be in failure mode, and takes corrective actions. Implementation requires that a monitoring process be created or used, and that all components being watched must be modified to handle the echo messages.

   b. Heartbeat: A component emits a heartbeat message at regular intervals and a monitoring component listens for it. If no heartbeat message is received within a predetermined time, the originating component is assumed to have failed, and corrective actions are taken. This tactic requires a monitoring component, and all components must be modified to send heartbeats at the proper intervals.

   c. Exceptions: Raise and handle exceptions. Exceptions are usually handled in the components in which they occur. Most modern programming languages include built-in support for exception handling. Implementation usually only requires minor, sub-architectural structural changes, such as the add of an exception handling block.

2. Fault Recovery – Preparation and Repair

   a. Voting: Processes running on redundant processors each take equivalent input and compute a single out value that is sent to a voter. The voter component decides which of the results is correct using an algorithm such as majority rules. The strongest approach is to implement each voting component independently; otherwise you can only detect hardware faults, and not algorithm faults. (If the voting components are running the same software, this tactic becomes very similar to Active Redundancy; see below.) To implement voting, create a voter component, and either replicate or write a new voting component.

   b. Active Redundancy: redundant components receive events in parallel, thus they are always in the same state. If one component fails, the other can immediately take over. This tactic that the processing component(s) to be replicated. It usually requires a central arbitrating component, although it is possible to make the redundant components perform the arbitrating without a central component.

   c. Passive Redundancy: One component is the active or primary component. It updates the state of one or more backup components. If the primary component fails, a backup component will be in approximately the same state, and will take over. This tactic requires that the primary component to be replicated to form the backup, and both are modified to implement the state update protocol between the primary and backup(s). A central arbitrating component may be needed.

d. Spare: A standby spare can replace different components, and is booted for the particular component that failed. This is most common where multiple components sharing load; imagine a system with multiple identical servers to service requests. Such a configuration is often called "N + 1 sparing." This requires the component of interest to be replicated, and changed to support the sparing. An important consideration is that all components must save state so the spare can replace any of them. A central arbitrating component may be needed.

3. Recovery – Reintroduction of a Failed Component

a. Shadow: When a component is restored, it runs in shadow mode behind an active component until its integrity is fully established. Implementation of this tactic requires a component to monitor the health of the "shadow" component. The shadow itself is probably replicated from another component.

b. State Resynchronization: Before a component is returned to service, synchronize its state with the current operation. The state synchronizing messages must come from somewhere; it is probably easiest to have them come from an active component; then a controlling component isn't needed.

c. Checkpoint/Rollback: Record consistent states and have a path to roll back to them if necessary. Each component in question must define its consistent states and implement a way to record the checkpoints and roll back to them. A component can usually do this without the need for a central component. However, note that the ease of implementation is based on how easy it is to define sane checkpoints – some systems have little notion of state.

4. Fault Prevention

a. Removal From Service: Remove a component from service to repair potential problems. For example, a component might be periodically restarted to prevent memory leaks from causing a failure. This can be implemented by modifying a component to restart itself, or by using a central monitoring component. If a central component is used, this tactic looks like a proactive process monitor; see below.

b. Transactions: Bundle actions into sets that can be undone all at once. Transactions are highly compatible with checkpoints as described above. Components are modified for transactions.

c. Process Monitor: When a fault is detected, a monitoring process manages the deletion of the process and its replacement. This is typically used together with the fault detection tactics of Ping/Echo or Heartbeat. Implementation requires a central monitoring component. The components being monitored may or may not need any changes.

7.2 Pattern and Tactic Interaction Data

We have organized the data by pattern. Within each, we give a short summary of the data for that pattern, followed by the descriptions of the tactics' impact.

The impact of each tactic is described as very positive, positive, neutral, negative, or very negative, as described earlier. They are abbreviated with the symbols + +, +, ~, -, - -, respectively. We show the Layers pattern as a sample.

Layers

Summary: The Layers pattern is very common in all systems. It provides good support for many fault tolerance tactics, and should be considered for highly reliable systems.

1. Fault Detection

a. Ping/Echo: +: The monitor component is added. If all the layers are implemented in a single process, then only the top layer must respond to the ping. If each layer is a separate process, then one approach is to create a hierarchy: the monitor pings the top layer, and each layer pings the layer below it before responding with an echo. (Add in the Pattern, Minor modifications)

b. Heartbeat: +: Similar to ping/echo, except the top layer sends the heartbeat based on time, rather than an echo in response to a ping. (Add in the pattern, minor modifications)

c. Exceptions: + +: Exceptions can be handled or propagated through layers with few if any changes to structure. (Implemented in the components)

2. Recovery – Preparation and Repair

a. Voting: +: Voting requires that the main processing components be rewritten to implement additional voting components. A new component, the voter, is added, and can be added as the top layer above all the voting components. Thus the pattern gives moderate support to this tactic. (Add in the Pattern, minor modifications)

b. Active Redundancy: +: To implement this tactic, replicate the layers without change. Add an additional layer above both layered systems to arbitrate and distribute messages to the redundant components. If it is a distributed system, this begins to look like a Broker. (Replicate, minor modifications, possibly add within the pattern)

c. Passive Redundancy: +: Replicate the layers, and then modify one layer to send and receive state updates (keep it to a single layer). It is likely the top layer, but may be a different layer. If needed, a monitor component can be added as a layer above both. This tactic does not fit quite as well as Active Redundancy, but is still a positive fit with Layers. (Replicate, minor modifications, possibly add within the pattern)

d. Spare: ~: One must replicate the layers, and modify some part of the replicated components to save their state. The spare must have a way to synchronize itself with the component that fails. A monitoring layer is probably needed.) In all this, the Layers pattern doesn't help, but it doesn't hurt, either. (Replicate, moderate modifications, add in the pattern)

3. Recovery – Reintroduction

a. Shadow: +: The system will have been duplicated, following one of the redundancy tactics. An additional layer is added to assess the health of the system coming back into service. If there is a monitoring layer for fault detection or redundancy, it will be the same layer. (Minor modifications, Add in the Pattern is in conjunction with another tactic.)

  b. State Resynchronization: +: Layers provides some support for states – state information can be encapsulated in a single layer, which can help with resynchronization of a component returning to service. A monitoring layer may be needed, and a layer is modified to send state information to the other component. The layers will have been replicated in another tactic. (Minor modifications, Add in the Pattern is in conjunction with another tactic.)

  c. Rollback: + +: Layers can provide checkpointing of data which can make it easier to recover the data or the state. A layer can hold a request to a lower layer that is pending. If the lower layer in unable to complete the transaction, the higher layer can readily undo the transaction. (Implemented in the components)

4. Prevention

  a. Removal from Service: ~: A monitoring component is needed. If all the layers are implemented in a single process, then the monitor is added as the top layer (Add in the pattern, little or no modification needed.) However, if each layer is a separate process, either the monitor must control each layer, or each layer must control the layer under it (both are significant modifications.) So depending on the implementation of Layers, it is either positive or negative.

  b. Transaction: + +: Layering encourages the packaging of actions into transactions; usually done at the highest layers. One can create commands to the

  c. Process Monitor: ~: Implementation depends on how the layers are implemented. If they are all in a single process, then simply add the process monitor as a top layer (Add in the pattern, little or no modification needed.) However, if each layer is a separate process, then either the process monitor must monitor each layer individually, or each layer becomes the process monitor of the layer under it. Both these cases require significant modifications to components. So it is either positive or negative, depending on the implementation of the Layers.

8. FUTURE WORK

This work has great potential to help software architects and designers be more effective in the design of reliable systems. But there is much to be done to make it truly useful.

8.1 Multiple Tactics, Patterns, and Quality Attributes

Most reliable systems use more than one fault tolerance tactic. This raises the question of how the implementations of multiple tactics influence each other. Because the patterns and tactics data are concerned with where in the system a tactic will be implemented, this data can also be used to help gain insight about the interactions of multiple tactics.

Our study of interactions of tactics is still rather preliminary. In particular, it appears that the really interesting interactions of tactics come between tactics from different quality attributes; for example tactics of fault tolerance and tactics of performance.

There are also interactions among patterns. We found that most systems incorporate at least two architecture patterns in their design. Where two or more architecture patterns are present, how does a tactic interact with them? While this is also preliminary work, we believe it works as follows: A tactic is implemented in one pattern and not the others. It makes sense, therefore, that the tactic is implemented in the pattern where it fits best. Of course, this is subject to the design of the application itself. For example, you might have a Shared Repository that has a Broker in front of it. If you implement Ping/Echo, the obvious choice for the Ping controller is the Broker, not the Shared Repository component.

The interaction of multiple quality attributes with respect to patterns and tactics requires study. While it is understood that quality attributes such as fault tolerance and performance interact, it is useful to examine the individual tactics and patterns involved for their interactions. This area is potentially highly complex and challenging.

8.2 Behavioral Considerations

The most visible part of architecture patterns and interactions with tactics is the structure of the two. While the preceding analysis includes behavior, the most obvious part is the structure. However, behavior also requires attention.

Each of the tactics introduces some additional behavior into the system. In this respect, one can consider a tactic to be a fault tolerance feature: behavior of the system with the goal of improving fault tolerance.

We noted that timing of actions is particularly important for certain fault tolerance tactics; for example:

- In Ping/Echo and Heartbeat, messages must be sent within a certain time period, or else the component is considered to be in failure.
- With Active and Passive Redundancy, messages must be sent to the redundant components within a certain timeframe; otherwise synchronization can be lost.

We did not study timing in detail, but it appears that timing may be an important issue with Pipes and Filters, where processing of data can be in large units.

We also noted that sequencing of actions is part of behavior. Both these aspects of behavior need to be studied in more detail to understand how they affect the patterns.

8.3 General Considerations

A general approach to using this data is warranted. We have explored using patterns to help create software architectures that satisfy quality attributes, and note that it can be used in published software architecture methods.

As this information grows, it should be organized and published for software designers. This will approach the concept of a handbook for software architecture.

9. REFERENCES

[1] Avgeriou, P. and Zdun, U. Architectural Patterns Revisited – a Pattern Language. In Proc. Of *10th European Conference*

on Pattern Languages of Programs (EuroPLoP 2005), (Irsee, Germany, July 6-10, 2005).

[2] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. E. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable and Secure Computing,* 1,1, Jan.-Mar. 2004, 11-33.

[3] Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice, 2nd edition*. SEI Series in Software Eng. Addison-Wesley Professional, Reading, MA, 2003.

[4] Bucchiarone, A., Muccini, H., and Pelliccione, P. 2007. Architecting Fault-tolerant Component-based Systems: from requirements to testing. *Electron. Notes Theor. Comput. Sci.* 168 (Feb. 2007), 77-90.

[5] Buschmann F. et al., *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, Chichester, England, 1996.

[6] de Lemos,R., Asterio de Castro Guerra, R., and Rubira, C. M. A Fault-Tolerant Architectural Approach for Dependable Systems, in *IEEE Software,* 23,2, March/April 2006, 80-87.

[7] Feng, Y., Huang, G., Zhu, Y., and Mei, H. 2005. Exception handling in component composition with the support of middleware. In *Proceedings of the 5th international Workshop on Software Engineering and Middleware* (Lisbon, Portugal, September 05 - 06, 2005). SEM '05. ACM, New York, NY, 90-97.

[8] Ferreira, G. R., Rubira, C. M., and Lemos, R. d. 2001. Explicit Representation of Exception Handling in the Development of Dependable Component-Based Systems. In *the 6th IEEE international Symposium on High-Assurance Systems Engineering: Special Topic: Impact of Networking* (October 24 - 26, 2001). HASE. IEEE Computer Society, Washington, DC, 182-193.

[9] Garcia, A. F., and Rubira, C. M. An Archietctural-based Reflective Approach to Incorporating Exception Handling into Dependable Software. In *Advances in Exception Handling Techniques*, Springer-Verlag, LNCS-2022, 2001, 189-206.

[10] Garcia, A. F., Rubira, C. M. F., Romanovsky, A. B., and Xu, J. A., A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software. In *Journal of Systems and Softawre* 59, 2, 2001, 197-222.

[11] Hanmer, R. *Patterns for Fault Tolerant Software,* Wiley, Chichester, England, 2007.

[12] Harrison, N. and Avgeriou, P. 2008. Analysis of Architecture Pattern Usage in Legacy System Architecture Documentation. In *Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008) - Volume 00* (February 18 - 21, 2008). WICSA. IEEE Computer Society, Washington, DC, 147-156.

[13] Harrison, N. and Avgeriou, P. Leveraging Architecture Patterns to Satisfy Quality Attributes, In proc. *First European Conference on Software Architecture*, Madrid, Sept 24-26, 2007, Springer LNCS.

[14] Harrison, N. and Avgeriou, P. 2007. Pattern-Driven Architectural Partitioning: Balancing Functional and Non-functional Requirements. In *Proceedings of the Second international Conference on Digital Telecommunications* (July 01 - 05, 2007). ICDT. IEEE Computer Society, Washington, DC, 21.

[15] Harrison, N., Avgeriou, P. and Zdun, U. Focus Group Report: Capturing Architectural Knowledge with Architectural Patterns, In proc. *11th European Conference on Pattern Languages of Programs (EuroPLoP 2006)*, Irsee, Germany.

[16] Harrison, N. and Meiners, J. H. 2006. The dynamics of changing dynamic memory allocation in a large-scale C++ application. In *Companion To the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA, October 22 - 26, 2006). OOPSLA '06. ACM, New York, NY, 866-873.

[17] Hofmeister, C.; Kruchten, P.; Nord, R.L.; Obbink, H.; Ran, A. & America, P. Generalizing a Model of Software Architecture Design from Five Industrial Approaches, In *Journal of Systems and Software*, 30,1, Elsevier, 2007, 106-126.

[18] Issarny, V. and Banatre, J. 2001. Architecture-based Exception Handling. In *Proceedings of the 34th Annual Hawaii international Conference on System Sciences (Hicss-34)-Volume 9 - Volume 9* (January 03 - 06, 2001).

[19] Laibinis, L. and Troubitsyna, E. 2004. Fault Tolerance in a Layered Architecture: A General Specification Pattern in B. In *Proceedings of the Software Engineering and Formal Methods, Second international Conference* (September 28 - 30, 2004). SEFM. IEEE Computer Society, Washington, DC, 346-355.

[20] Magee, J. and Maibaum, T. 2006. Towards specification, modelling and analysis of fault tolerance in self managed systems. In *Proceedings of the 2006 international Workshop on Self-Adaptation and Self-Managing Systems* (Shanghai, China, May 21 - 22, 2006). SEAMS '06. ACM, New York, NY, 30-36.

[21] Muccini, H., Pelliccione, P., and Romanovsky, A. 2007. Architecting Fault Tolerant Systems. In *Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture* (January 06 - 09, 2007). WICSA. IEEE Computer Society, Washington, DC, 43.

[22] Noble, J., and Wier, C. *Small Memory Software: Patterns for Systems with Limited Memory*, Addison-Wesley, Reading, MA, 2001.

[23] Shaw, M. and Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline.* Addison-Wesley, Reading, MA, 1996.

[24] Utas, G. *Robust communications Software: Extreme Availability, Reliability and Scalability for Carrier-Grade Systems,* Wiley, Chichester, England , 2005.