Author(s)	Alho, Pekka; Rauhamäki, Jari
Title	Patterns for Light-Weight Fault Tolerance and Decoupled Design in Distributed Control Systems
Citation	Alho, Pekka; Rauhamäki, Jari 2013. Patterns for Light-Weight Fault Tolerance and Decoupled Design in Distributed Control Systems In: Eloranta, Veli-Pekka; Koskinen, Johannes; Leppänen, Marko (ed.) . Proceedings of VikingPLoP 2013 Conference Ikaalinen, Finland 21.3. - 24.3.2013. Tampere University of Technology. Department of Pervasive Computing. Report vol. 2, Tampere, 1-17.
Year	2013
Version	Post-print
URN	http://URN.fi/URN:NBN:fi:tty-201403261137

Patterns for Light-Weight Fault Tolerance and Decoupled Design in Distributed Control Systems

Pekka Alho[1], Jari Rauhamäki[2]

[1] Tampere University of Technology, Dept. of Intelligent Hydraulics and Automation, Finland
pekka.alho@tut.fi
[2] Tampere University of Technology, Dept. of Automation Science and Engineering, Finland
jari.rauhamaki@tut.fi

1 Introduction

Distributed control systems are continuously gaining importance, as more and more devices and machines are equipped with embedded systems that control their operation. These controllers are increasingly more powerful and networked, providing intelligence and interoperability for the control system. Examples of such systems range from large mobile machines to robots and intelligent sensor networks. These systems often interact with physical processes, influencing many parts of our lives either directly or indirectly. Therefore they need to be *dependable*, which can be measured with the attributes of availability, reliability, safety, integrity and maintainability [1]. However, with the increased functionality and intelligence, the complexity of these systems is also increased, meaning that the development process becomes more demanding and dependability becomes more costly to achieve and verify. Another significant requirement of these systems is that they usually are real-time systems, which may put limitations on the architecture.

Many critical systems that have failed catastrophically are well-known – examples such as Therac-25 radiation therapy machine and the explosion of Ariane 5 rocket are infamous, whereas highly reliable systems receive little recognition, even though their study might give valuable ideas for the design and architecture of new software. One example of such systems can be found in telephony applications, namely Ericsson AXD301 ATM switches that achieved nine nines (99.9999999%) service availability, running software written in Erlang [2]. Erlang's highly decoupled actor model and fault handling based on supervisors have inspired especially LET IT CRASH and SERVICE MANAGER patterns found in this paper.

This paper presents three software patterns that can be used to improve control system dependability by implementing a decoupled architectural design with supporting fault handling. The decoupled architecture can also be used to introduce additional fault tolerance solutions – like checkpointing and rejuvenation – gradually to the system, until a sufficient level of reliability has been achieved [3]. Our patterns have been encountered originally in research of remote handling control systems used to teleoperate robotic manipulators, but all patterns have examples of other known uses as well. Some of these examples are presented in the corresponding sections of the patterns.

2 Context of the Patterns

Fault tolerance cannot be implemented without redundancy of some kind. To have fault tolerance for e.g. computer failures, we would need at least two computers – if one fails the other one can detect the error and try to correct it. Software faults on the other hand are typically development faults, which are harder to detect and correct than hardware faults. To have good coverage for software faults, we would need diverse redundancy, but even this form of fault tolerance has been criticized of being susceptible to common mode failures [4]. Development costs for design diversity are also often seen as prohibitive.

The patterns in this paper present an alternative approach to fault tolerance, based on dividing the system into highly decoupled modules and using this to implement lightweight form of fault tolerance. We present an architectural pattern for this called DATA-CENTRIC ARCHITECTURE but this is of course not the only way to achieve a high level of decoupling. One of the key points of decoupling is that it should by itself improve reliability by limiting fault propagation and improving modularity and understandability of the system. In a way, modular approach can be seen like compartmentalization of ships – without compartments, every leak can sink the ship. An example of a software system that uses modularity to successfully implement fault isolation and resilience is the MINIX operating system, based on the idea of microkernel [5].

Modular and decoupled architecture can also be used to implement other reliability-improving patterns like SERVICE MANAGER and LET IT CRASH documented in this paper or other well-known patterns like LEAKY BUCKET COUNTER [6], WATCHDOG [6] [7], etc. The patlets of the patterns presented in this paper are listed in the **Table 1**. List of all referenced patterns with short descriptions can be found in an appendix.

Table 1. Patlets

Pattern	**Description**
DATA-CENTRIC ARCHITECTURE	How to implement reliable and scalable distributed control system? Use data-centric middleware based on PUBLISH/SUBSCRIBE model [8] to reduce level of coupling between modules.
SERVICE MANAGER	How to detect faults and restart modules or processes after a failure? Implement a service manager that can monitor, start and stop modules.
LET IT CRASH	How to react to failures without crashing the whole system? Flush the corrupted state by “crashing” the process instead of writing extensive error handling code. Let some other process like service manager do the error recovery e.g. by restarting the crashed process.

The presented patterns work together by building on the top of features provided by the higher abstraction level patterns as shown in **Fig. 1**, but all of the patterns are also typically used separately and in contexts other than distributed control systems.

The DATA-CENTRIC ARCHITECTURE provides the decoupled architectural model needed to use LET IT CRASH for fault handling. The SERVICE MANAGER pattern provides a way for trying recovery after failures, in addition to providing error detection and monitoring.

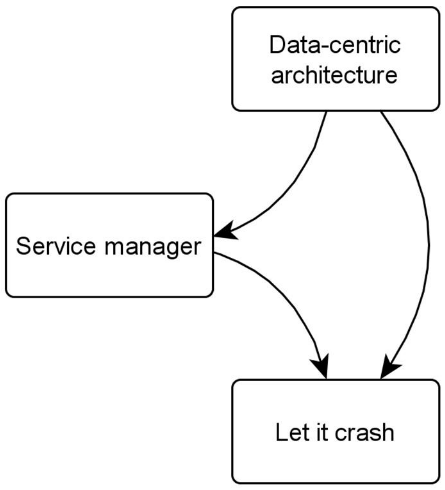
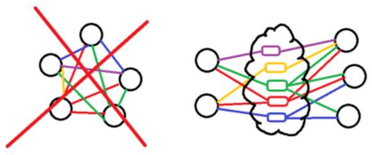


Fig. 1. Graph of pattern relationships

The idea of crashing a process suggested by LET IT CRASH may sound like a risky action to take. However, the idea is to offer recovery from transient physical and interaction faults (sometimes called Heisenbugs), ability to keep the system as a whole functioning, even if some internal process would crash, and possibility to hot-swap code and bug-fixes. The downside of this approach is of course that it is not suited for fail-operate systems like flight controllers that must be operational all the time – this type of systems would be the right domain to apply design diversity.

3 Patterns

3.1 Data-Centric Architecture



Intent. Implement an architecture based on decoupled modules (e.g. services, components or processes) that are connected with data-centric middleware.

Context. You are developing a distributed control system that consists of several subsystems and needs to interact with other heterogeneous systems like mobile machines or plant systems. The system has CPU and memory resources available to run an operating system – rather than being based on a basic time-triggered scheduler used in resource-constrained embedded systems. Failures in control functions (e.g. boom or manipulator control) may cause damage to the environment and equipment, meaning that some subsystems may be categorized as safety-critical.

Problem. How to implement a reliable and scalable distributed control system?

Forces.

- *Throughput*: Some time-critical data like sensor measurements may be updated with short period, producing large amounts of communication.
- *Scalability*: New nodes and subsystems can join the system any time; assumptions about interfaces between modules should be minimized.
- *Changeability*: System configuration and functionality might change. Changing interfaces in a tightly coupled system requires code changes at both ends (and at all clients), so assumptions about expected behavior should be minimized. Point-to-point protocol based client-server architectures (like sockets or remote method invocation) are not ideal because of complexity and coupling introduced.
- *Maintainability and long expected life-cycle*: The control system has long expected lifetime and needs to be maintainable and extensible in the future – if subsystems are added or substituted, changes to existing modules need to be minimized. System should be easy to understand and modify without breaking it.
- *Reusability*: Same modules could be used in other control system implementations.
- *Interoperability*: Distributed control systems consist of and/or need to communicate with heterogeneous platforms.
- *Testability*: Tightly coupled modules are difficult to test because they are more dependent on other modules.
- *Availability*: The system as a whole should remain available, even if some subsystems or processes experience failures.
- *Reliability*: A single fault in the control system software should not endanger functionality of the whole system (i.e. no single point of failures).
- *Real-time performance*: Control system interacts with the real world and needs to react in a deterministic manner.
- *Safety*: Need to detect if a module has crashed or is down (not releasing new information) so that the system can enter SAFE STATE [7] in a controlled fashion. Safety-critical and non-safety-critical subsystems cannot be tightly coupled, since errors may propagate.

- *Quality of service*: Different subsystems may have different requirements for quality of service[1] (QoS) policies. There is an impedance mismatch between e.g. real-time control systems that operate on a timescale of milliseconds and enterprise/high level systems that are several orders of magnitude slower.

Solution. Use data-centric middleware based on PUBLISH/SUBSCRIBE model to reduce level of coupling between modules.

Implement data exchange between modules by adopting a middleware that publishes data to a global data space instead of sending point-to-point messages or remote procedure calls; data-centric architecture is based on removing direct inter-module references by exposing the data and hiding the code. Management of the global data space is externalized to the middleware that implements a topic-based PUBLISH/SUBSCRIBE model. The middleware acts as a single source of up-to-date state information in the system, instead of applications managing state separately.

Modules do not need to know recipients of the data when publishing it, which reduces coupling. Instead of sending data directly to a recipient, it is published to a topic. Data can be e.g. sensor measurements, events or commands, but it must follow a shared data model which is represented as topics in the actual system implementation. Publishers register as data writers to a topic and interested subscribers can join the topic as data readers. The middleware automatically discovers new readers and writers, which means that new nodes can join the system on the fly. Single topic can have multiple readers and writers, as shown in **Fig. 2**. Moreover, a topic can have multiple instances, which are identified by a key value.

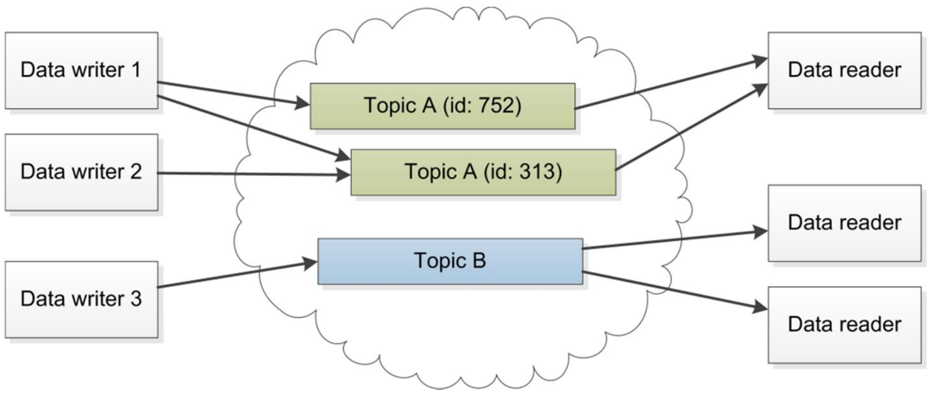


Fig. 2. Data is published to topics that can have multiple data writers and readers. Topic A has two instances, identified by the id number key value.

Expose data and hide the behavior. Instead of designing interfaces for components, you must design how to represent the state of the system and the external or internal

[1] QoS policies provide the ability to specify various parameters like rate of publication, rate of subscription, reliability, data lifespan, transport priority, etc. to control end-to-end connection properties. Policies can be matched on a request vs. offered basis.

events that can affect it. This needs a common data model, which captures the essential elements of the physical system and application logic. Conceptually the data model is similar to class diagram in object-oriented programming, since it consists of identifying entity types, which have data attributes assigned to them, and associations. The difference is that the data model focuses exclusively on data and not behavior.

Separate communication and application logic. Delegate network communications to a "data bus" formed by the publish/subscribe middleware (**Fig. 3**), so that the application logic can focus on the core functionality. Middleware takes care of maintaining the data up-to-date, automatically updating new nodes that join. If the middleware uses a central server as a message BROKER [8], it becomes a single-point-of-failure and possibly a bottleneck. Therefore, choose a decentralized middleware solution if possible to avoid this problem

Define appropriate QoS attributes for the data topics (reliability, durability, deadlines, etc.). Middleware manages the data lifecycle according to the associated QoS policy and matches policies offered by publishers vs. policies requested by subscribers.

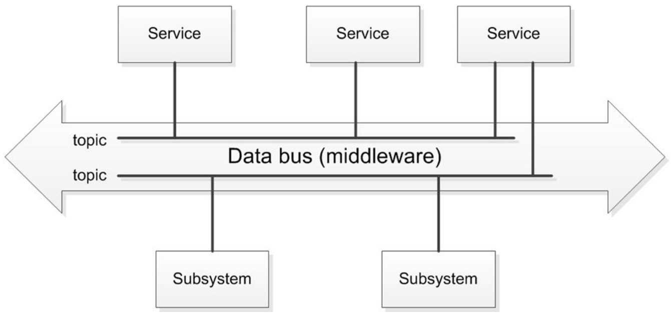


Fig. 3. Middleware implementation as a data bus that has no central components or brokers. Services and subsystems can join topics as publishers and/or subscribers.

Choose module granularity (size of the communicating modules) so that performance is not compromised. On the other hand, too large modules size may diminish the benefits of the data-centric architecture. The communication participants can be e.g. subsystems, applications, processes or modules, depending on the environment. Note that communicating modules can also exist on a same computer and use the data bus to get benefits of loose coupling locally.

Compared to message-centric publish/subscribe, the difference in data-centric model is that middleware "messages" – i.e. topic samples – are transparent to the middleware. In message-centric model, middleware does not know or care about message contents. With smart, QoS-aware data-centric middleware, application components can be leaner and take less time to develop because the logic that implements the QoS functionality is pushed down into the middleware. The application specifies these policies to the middleware during launch and it gets notified by the middleware during operation when QoS requirements are not being met.

Consequences.
+ Publishers do not need to know about subscribers.
+ Middleware provides interoperability between heterogeneous platforms
+ Decoupled design provides error confinement and other benefits like improved maintainability.
+ Modules can be changed dynamically because late joiners receive new data automatically; ability to hot-swap code can be easily implemented.
+ Network transport layer is abstracted as communications are externalized to middleware, which reduces communication related code and simplifies implementation.
+ Gives developers control of data delivery with QoS management; QoS can be used e.g. to guarantee reliable delivery eventually or that available data is kept up-to-date with best effort. Former would be useful for sending status changes or commands whereas latter could be used for sensor measurement for which guaranteeing delivery of old information makes no sense.
+ Reusability is improved since modules are using shared memory and have their own namespaces, etc.
+ Publish/subscribe based middleware scales effectively since recipients for data are not explicitly defined.
+ Performance gains can be achieved on multi-core machines since modules can be easily parallelized and they communicate asynchronously.
+/- Needs good and consistent data-models that must be managed and maintained.
- Sending of commands is not as straightforward as in client-server architectures since commands need to be parsed from the data. Serialization and deserialization of the data structures for transmission may also add overhead.
- Parsing of data complicates debugging because it adds another potential source for faults. If data is parsed incorrectly, it may not be self-evident where the fault originates.
- Errors in the middleware itself might complicate testing and be hard to detect.
- Middleware solution adds some overhead to message size and uses system resources.
- Possible vendor lock-in to the middleware provider.

Examples. Data Distribution Service for Real-Time Systems (DDS) is decentralized and data-centric middleware based on the publish/subscribe model. DDS is aimed at mission-critical and embedded systems that have strict performance and reliability requirements. Therefore, its implementations have typically been optimized and tested to suit the needs of these systems. DDS is used as the information backbone in the Thales TACTICOS naval combat management system that integrates various subsystems like weapons, sensors, counter measures, communication, navigation, etc. to a “system of systems”. Applications are distributed dynamically over a pool of computers in order to provide combat survivability and avoid single-point-of-failures. System configuration can be adapted for use in various mission configurations, on-board & simulator training, and different ship types.

Related Patterns. BUS ABSTRACTION [7], and PUBLISHER-SUBSCRIBER.

MEDIATOR [9] increases decoupling in a similar fashion, but is designed to decrease connections between objects locally.

3.2 Service Manager

Also Known as. SUPERVISOR or SERVICE GATEWAY.

Intent. Service manager starts, stops, and monitors processes locally and takes care of resource allocation for systems that need high availability and real-time performance.

Context. You are developing a system with highly decoupled architecture (e.g. using DATA-CENTRIC ARCHITECTURE) that consists of large number of processes or tasks (services). These processes have dependencies and therefore need to be started in specific order. Process composition may change dynamically during runtime because your system will have intelligent functionality, it needs to adapt to new situations, or different functionalities need to be tested without stopping/restarting the whole system.

You know rough upper-limit estimates for how much system resources like memory and CPU time the processes will use.

The system has long expected life-cycle. It is likely to be deployed on a remote location like a forest or a control cubicle, making direct physical interaction with the system a bothersome task.

If you have a real-time operating system and a task gets stuck in a while loop or some other control structure, it freezes the whole system as other lower priority processes (including input devices and network connections) cannot get CPU time. In this case, the only option is usually to restart the whole computer manually.

Problem. How to ensure that all dynamic modules in your control system are running correctly and you have enough system resources to achieve deterministic real-time performance?

Forces.

- *Availability*: The system as a whole should remain available, even if some subsystems or processes experience failures, in order to able to use other parts of the sys-

tem that are not connected to the failed subsystem. The system must detect faults and try to recover from them automatically. If a failure needs immediate reaction from a human operator, the system will not scale cost-efficiently and reliably.
- *Data logging/testability*: If a process fails, the failure should be detected and logged.
- *Real-time performance*: The control system needs to respond in a deterministic and predicable manner. Predictability includes system behavior when a fault is triggered.
- *System resources*: Control systems are typically deployed on embedded devices that have limited memory and CPU resources available. They may need to be monitored in order to guarantee the real-time performance of the system.

Solution. Implement a service manager that can monitor, start and stop modules. Create a local parent process (the service manager) that is responsible for starting, stopping and monitoring its child processes. The basic idea of the service manager is to keep its child processes alive by restarting them when necessary. Location of the service manager is on the same computer as the child processes in order to keep implementation simple. Therefore, all computers in the system need their own, independently functioning, service managers. The service manager is given the highest process priority in the system or put in the kernel so that a faulty real-time process cannot prevent it from functioning by consuming all available CPU time.

Start the child processes based on a fixed order or a dependency table read from a configuration file, similar to START-UP MONITOR [7], and/or implement a user interface that can be used to start and stop processes.

Use the service manager to allocate resources like CPU time and memory for the child processes and monitor their use. Expected maximum resource consumption can be specified in the same configuration file that is used for starting services. New processes are not started if there are not enough resources available. If a process consumes more resources than expected, it can be restarted, leading to error handling according to the LET IT CRASH pattern. Resource use can be followed e.g. with proc filesystem or *getrusage* call in Unix-like systems.

Since the key functionality of service manager is to monitor processes for failures, error detection can be based on additional or alternative techniques besides resource monitoring. This can be done with e.g. operating system features, HEARTBEAT [6] [7] or WATCHDOG.

If the service manager is deployed on a system that uses DATA-CENTRIC ARCHITECTURE, service startup interfaces can be implemented through the middleware. Since the middleware abstracts the location of the data, it can be used to remotely start dependencies. Example: service manager SM_A must start a service called S1. However, it has a dependency called S2 which cannot be found locally, so the service manager publishes a start request for S2. A second service manager SM_B on another computer notices the request, starts S2 and publishes information about the successful startup. SM_A receives information that S2 is available and starts S1.

The implementation for service manager needs to be kept fairly simple, since it acts as a single point of failure locally. This conflicts with the need to use of configuration files, making resource checks, and providing user interface, so they should be based on external components or libraries that have been proven in use.

Consequences.

+ Detects and initializes recovery from transient faults that cause a process to consume too much system resources or become unresponsive. If the fault is persistent, LEAKY BUCKET COUNTER can be used to limit the number of restarts.

+ Ensures other processes stay alive and have sufficient resources.

+ Simplifies starting procedure of complex system that consists of large number of processes, making possible to start and stop a large number of processes automatically and in a specific order.

+ Cost-efficiency: the same service manager implementation can be reused on several systems.

+ Supports logging and reporting of errors so that they do not go undetected.

- Cannot detect faults that cause erroneous output for monitored components.

- Cannot recover persistent faults like development and physical faults, e.g. computer failures.

- Potential single point of failure that may stop the entire system from working if services are incorrectly terminated.

- Restarting a service may cause the system to behave in non-deterministic way and miss deadlines, which is a failure for a hard real-time system. However, it should be noted that the failure would have likely cause the system to miss the deadlines or exhibit some other unwanted behavior in the first place.

- Resource utilization needs to be estimated for the processes in order to set limits.

- Service manager uses system resources and may reduce performance.

Examples. The MINIX, a POSIX conformant operating system, based on a microkernel that has minimal amount of software executing in the kernel mode. The rest of the operating system runs as a number of independent processes in user mode, including processes for the file system, process manager, and each device driver. The system uses a special component known as the driver manager to monitor and control all services and drivers in the system [5]. Driver manager is the parent process for all components, so it can detect their crashes (based on POSIX signals). Additionally the driver manager can check the status of selected drivers periodically using HEARTBEAT messages. When a failure is detected, the driver manager automatically replaces the malfunctioning component with a fresh copy without needing to reboot the computer. The driver manager can also be explicitly instructed to replace a malfunctioning component with a new one.

Open source tool Monit (http://mmonit.com/monit/) can function as a service manager in non-real time systems. Following code listing shows an example configuration for Spamassassin daemon that restarts the daemon if its memory or CPU usage exceeds 50% for 5 monitoring cycles:

```
check process spamd with pidfile /var/run/spamd.pid
   start program = "/etc/init.d/spamd start"
   stop  program = "/etc/init.d/spamd stop"
   if 5 restarts within 5 cycles then timeout
   if cpu usage > 50% for 5 cycles then restart
   if mem usage > 50% for 5 cycles then restart
   depends on spamd_bin
   depends on spamd_rc
```

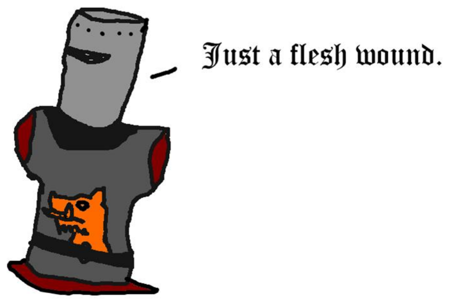
Related Patterns. FAULT OBSERVER [6], HEARTBEAT, SAFE STATE, SOMEONE IN CHARGE [6], START-UP MONITOR, STATIC RESOURCE ALLOCATION [7], and WATCHDOG.

To see how to design an application in a way that it can be easily restarted at any time, see LET IT CRASH.

MANAGER design pattern [10] can be used to manage multiple objects of same type – the idea is similar to SERVICE MANAGER (keep track of entities and provide unified interface for them) but the MANAGER focuses on different scope, i.e. managing entities (objects) of the same type and does not include resource monitoring or fault detection.

SYSTEM MONITOR [6] can be used to study behavior of system or specific tasks and make sure they operate correctly, e.g. by using HEARTBEAT or WATCHDOG. If a monitored task stops, SYSTEM MONITOR reports the error. Compared to it, SERVICE MANAGER has a more active role in managing the tasks.

3.3 Let It Crash



Also Known as. CRASH-ONLY [11], FAIL-FAST, LET IT FAIL or OFFENSIVE PROGRAMMING.

Intent. Avoid complex error handling for unspecified errors. Instead, crash the process and leave error handling for other processes in order to build a robust system that handles errors internally and does not go down as a whole.

Context. You are developing a distributed control system that consists of several processes and subsystems that need to cooperate to complete tasks.

DATA-CENTRIC ARCHITECTURE or some other asynchronous decoupled architectural design has been utilized so that processes are not using shared memory.

Some subsystems might have safety-critical functionality, but it is possible to move the system to SAFE STATE (i.e. the system is fail-safe type, not fail-operate). The system has dynamic state information from the user inputs and working environment in the process memory, e.g. tool tracking data in the case of a robot manipulator. This state data needs to be recovered after a failure.

The system has a mechanism like monitoring layer, supervisors or a restart manager for restarting the processes. This can be implemented at operating system, programming language or framework level, e.g. with the SERVICE MANAGER.

Problem. How to implement lightweight form of error handling that improves reliability and predictability?

Forces.

- *Availability*: The system as a whole should remain available, even if some subsystems or processes experience failures, since degraded functionality is better than no functionality. In case of a fault, only minimal part of the system should be affected. Recovery from failures should happen without human intervention and with minimal downtime.
- *Reliability:* Generation of incorrect outputs should be prevented, otherwise errors may propagate and the system could cause damage to the environment.
- *Safety*: If an error is detected, any functionality using the affected process should be stopped and taken to a safe state in order to prevent and minimize damages.
- *Cost-efficiency*: Design diverse fault tolerance techniques are oversized or impractical for the application, but the system needs to be able to recover from errors.
- *Real-time performance*: Control system needs to react within a certain time-limit; exceeding the time-limit causes a failure.
- *Predictability*: The system should behave in a consistent manner. If the process tries to repair its corrupted state, behavior of the system cannot be predicted, which complicates debugging and verification of reliability. Predictability includes system behavior when a fault is triggered.
- *Error handling*: Because it is impossible to foresee all possible faults, specifications do not cover all possible error situations. Error situations occur seldom, are difficult to handle and non-trivial to simulate in testing [11]. If the programmers try to implement error handling, they will make ad hoc decisions not based on the specifications (i.e. they cannot know how the error should be handled), possibly causing unwanted and undocumented behavior.

Solution. Make processes crash-safe and fast to recover; flush corrupted state by “crashing” the process instead of writing extensive error handling code.

Commodore 64, DOS machines and other old computers were designed to be shut down by simply turning the power off, essentially crashing the system. On the other hand, if an operating system caches disk data in memory, workstation crash may corrupt the file system, which is inconvenient and slow to repair. Control system processes and subsystems should also be designed to be easily terminated and recoverable with a simple recovery path if an error is detected, instead of guessing how error recovery should be attempted, possibly corrupting program state further and causing unpredictable behavior.

Therefore, implement error handling by terminating the process that has encountered the error. Only program extended error recovery routines if they are based on the specification or it is self-evident how the error should be handled – otherwise crash the process. However, only the module or process where the error is should be crashed, not the whole system.

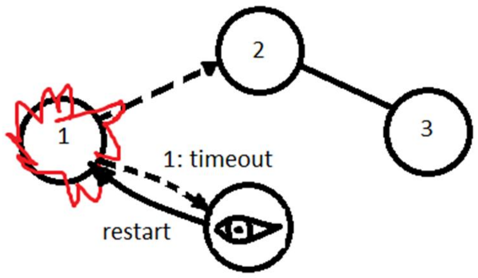


Fig. 4. Process 1 encounters an error and dies, after which it is restarted by the service manager, represented as an eye. If the process 2 detects a deadline overrun, it needs to stop, potentially interrupting process 3, and wait until process 1 is active again before resuming work. Alternatively the process 2 does not notice any deadline overruns and continues working normally.

Processes that have been designed with LET IT CRASH can 1) help to find faults, by making them more visible ("offensive programming"), and 2) be used to implement fault tolerance (recovery from faults or software rejuvenation). In latter case it is possible to do recovery without affecting service availability if the recovery process is fast enough. Recovery (and rejuvenation) also needs an external entity to initiate it, since the process itself has crashed (see **Fig. 4**). This pattern focuses mostly on the second case since it is more problematic to implement correctly.

You have a monitoring layer that can recover the system (e.g. by restarting). However, to detect a failure, the failed application or system service may first have to die. In this case the process terminates itself immediately upon encountering an error. Abnormal program termination can be forced e.g. by using *abort()* or *raise(SIGSEGV)*. If the monitoring layer has implemented failure detection – based on watchdog, heartbeat, etc. – it can also hard-fail the service using e.g. *kill(pid, SIGTERM)*.

Error recovery is performed by restarting the process. Therefore, make processes fast and easy to restart in order to minimize service failures and downtime. To keep recovery path simple, use the single responsibility principle, thereby minimizing responsibilities of a single process. If the process encounters an error and crashes, it might be possible to recover from the error without causing deadline misses for other processes and tripping the system to a SAFE STATE. However, if a control loop has a period of e.g. 1 ms and restarting of a process that provides information for the loop takes several milliseconds, control loop execution will be interrupted temporarily.

Recovery paths can be tested extensively by terminating the system forcibly every time it needs to be shut down or restarted, instead of letting it run through a normal shutdown process. This forces the system to do a recovery during the startup

Make processes crash-safe. Processes typically handle three types of state data: dynamic, static, and internal. Internal state is related to current computations and is usually discarded after use. If a process crashes, you must think if you want to recycle its internal state. If you recycle everything you risk hitting the exact same fault again and crashing, so it might be reasonable to recycle only parts of this state. Static state is configuration data that can be easily recovered or read from other processes. Finally, the dynamic state data is generated as the program is executed by reading user inputs, interacting with other processes and environment, etc. Some of it can be computed from other data or read directly from sensors, but the critical problem is the data from user or environment that cannot be reconstructed. This data must be protected by using checkpointing, journaling or some other form of dedicated state store like databases and distributed data structures.

Implement a reporting functionality that reports failures so that they do not go unnoticed. Failure information can be forwarded e.g. by using a service manager or supervisors to send NOTIFICATION messages [12].

The corollary to the LET IT CRASH approach is that you must design your software to be ready for processes failing. There is now a possibility that a dependency is not available because it has been crashed and is being restarted. To detect this situation, add timeouts or appropriate QoS policies to interactions between components. If the timeout is triggered, move the system to a SAFE STATE. Normal operation can be resumed when dependencies are back online. A missing dependency is therefore not considered to be an error that would necessitate a crash.

Consequences.
+ Enables simple error handling & recovery; avoids complex error handling constructs in code, therefore improving predictability of the system.
+ Cost-effective (lightweight) form of fault tolerance that does not require use of redundancy.
+ Allows error handling to be implemented separately (externally) from the business logic, e.g. with supervisors.
+ Supports recovery from transient faults since a restart is usually enough to handle them.
+ Possible to achieve high availability (for the system as a whole, not necessary for all services provided by the system).
+ Compatible with other fault tolerant designs like redundancy.

+ Processes can be updated to new versions on-the-fly, since the old process can be killed and replaced using the normal recovery path.
+ Limits error propagation to other parts of the system (babbling idiot failure) by acting as an ERROR CONTAINMENT BARRIER [6].
+ Errors are less likely to cause the system to perform unpredictable and potentially dangerous or irreversible operations.
+ Finding faults should be easier, since they are made more visible by crashing and reporting.
- Availability of some services provided by the system is lower (when compared to redundant fault tolerance solutions) – on the other hand availability of other unrelated services provided by the system should be unaffected.
- Cannot mitigate persistent faults.
- Processes need additional code to react to missing dependencies (i.e. other services, when waiting for them to come back online).
- Possible performance cost if state needs to be saved to enable recovery.
- Recovery speed is non-deterministic since it depends on how fast the processes can be restarted, loading of saved state, loading of dependencies, system load level, etc.

Examples. Erlang actor model and supervisors (Erlang is used e.g. in Ericsson AXD301 ATM switches) [2]. Supervisors are processes that are responsible for starting, stopping and monitoring their child processes. The basic idea of a supervisor is that it should keep its child processes alive by restarting them when necessary [13].

Related Patterns. ERROR CONTAINMENT BARRIER, NOTIFICATIONS, SAFE STATE, SERVICE MANAGER, REDUNDANCY [6].

Software REJUVENATION [11][14] is a proactive technique where the system has been designed to be booted periodically. Microrebooting [11] refers to a technique where suspect components are restarted before they fail.

MINIMIZE HUMAN INTERVENTION (MHI) is about how the system can process and resolve errors automatically before they become failures [6]. LET IT FAIL could be implemented as part of MHI as a final resort or instead of MHI in case there is no specification for error handling.

References

1. Avizienis, A., Laprie, J.-C., Randell, B., & Landwehr, C. (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. Transactions on Dependable and Secure Computing, 1(1).
2. Armstrong, J. (2003). Making Reliable Distributed Systems in the Presence of Software Errors. Stockholm, Sweden: Royal Institute of Technology.
3. Dunn, W. (2002). Practical Design of Safety-Critical Computer Systems. Reliability Press.
4. Knight, J., & Leveson, N. (1986). An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming. Transactions on Software Engineering, 12, 96-109.

5. Herder, J. (2010). Building a Dependable Operating System: Fault Tolerance in MINIX 3. Netherlands: Vrije Universiteit. USENIX Association.
6. Hanmer, R. (2007). Patterns for Fault Tolerant Software. John Wiley & Sons.
7. Eloranta, V.-P., Koskinen, J., Leppänen, M., & Reijonen, V. (2010). A Pattern Language for Distributed Machine Control Systems. Tampere University of Technology, Department of Software Systems.
8. Buschmann, F., Henney, K., & Schmidt, D. (2007). Pattern Oriented Software Architecture: A Pattern Language for Distributed Computing. John Wiley & Sons.
9. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
10. EventHelix.com Inc. Manager Design Pattern. Retrieved January 2, 2013, from EventHelix: http://www.eventhelix.com/realtimemantra/ManagerDesignPattern.htm#.UOQm6kUbR8E
11. Candea, G. & Fox, A. (2003). Crash-Only Software. Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems.
12. Eloranta, V.-P. (2012). Event Notification Patterns for Distributed Machine Control Systems. Proceedings of VikingPLoP 2012 Conference. Tampere University of Technology, Department of Software Systems.
13. Erlang/OTP R16A documentation. Retrieved February 13, 2013, from: http://www.erlang.org/doc/
14. Hanmer, R. (2010). Software Rejuvenation. Proceedings of 17th Conference on Pattern Languages of Programs. ACM.

Acknowledgements. Authors would like to thank Robert Hanmer for providing valuable comments during the shepherding process and VikingPLoP 2013 participants for the feedback. This work was carried out under the EFDA Goal Oriented Training Programme (WP10-GOT-GOTRH) and financial support of TEKES, which are greatly acknowledged. The views and opinions expressed herein do not necessarily reflect those of the European Commission.

Appendix: List of Referenced Patterns

Table 2. Short descriptions of referenced patterns.

Pattern	**Pattern intent**
BUS ABSTRACTION [7]	Nodes communicate via a message bus. The bus is abstracted so it can be changed easily.
ERROR CONTAINMENT BARRIER [6]	System should stop the flow of errors from one part to another by isolating them to a unit of mitigation and initiating error recovery.
FAULT OBSERVER [6]	Coordinate reporting to all observers that a fault is present, reported, and recovery actions escalated.
HEARTBEAT [6] [7]	Send a status report at regular intervals to let other parts of the system know their status.
LEAKY BUCKET	Implement a method to ride over transients by keeping a counter

COUNTER [6]	that is automatically decremented and incremented by errors.
MONITOR [10]	Support many entities of same or similar type. The MANAGER object is designed to keep track of all the entities. In many cases, the MANAGER will also route messages to individual entities.
MEDIATOR [9]	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
NOTIFICATIONS [12]	Communicate noteworthy or alarming events and state changes in the system using a dedicated message type.
PUBLISH/SUBSCRIBE [8]	Define a change propagation infrastructure that allows publishers in a distributed application to disseminate events that convey information that may be of interest to others. Notify subscribers interested in those events whenever such information is published.
REDUNDANCY [6]	Maximize availability by having alternate hardware or software that can perform the same function.
REJUVENATION [11][14]	Periodically rejuvenate a software item by shutting it down and restarting it.
SAFE STATE [7]	If something potentially harmful occurs, all nodes should enter a predetermined safe state.
SOMEONE IN CHARGE [6]	Every fault tolerance action undertaken by the system should have a clearly identified entity controlling and monitoring the action.
START-UP MONITOR [7]	During start-up all devices are started in certain order and with correct delays. Additionally, care is taken that there are no malfunctions.
STATIC RESOURCE ALLOCATION [7]	Critical services are always available when all resources are allocated when the system starts.
SYSTEM MONITOR [6]	Some errors will only manifest themselves at a system level. Check for them at this level.
WATCHDOG [6] [7]	Build a special entity to watch over another to make sure that it is still operating well.