2.5 – Strategies for Achieving Robustness in Coalitions of Systems

Mary Shaw
School of Computer Science
Carnegie Mellon University
mary.shaw@cs.cmu.edu

November 2006

"Robustness" is an overarching property of software systems that includes, to various viewers and to various extents, elements of correctness, reliability, fault-tolerance, performance, security, usability (without surprises), accuracy, and numerous other properties. Robustness is a form of dependability that focuses on resilience to failures.

Many aspects of dependability and robustness have been explored extensively in the context of individual components. Modern software systems, however, are composed from multiple components. Often these components have not been designed to operate together. Increasingly these components are legacy code or even applications that can operate alone as well as in concert. Further, the components may be data or services as well as code. The challenge of individual components lies in understanding and managing the code, but the major challenge of modern systems lies in understanding and managing the interactions among the components. Large-scale system integration encounters new sources of problems, such as architectural mismatch, cross-platform portability, and side effects of evolution of the computing infrastructure.

This new setting qualitatively changes the nature of the software development and integration process.

Classical software	***Modern systems***
Localized	Distributed
Independent	Interdependent
Insular	Vulnerable
Installations	Communities
Centrally-administered	User-managed
Software	Information resource
Systems	Coalitions

In this setting, "coalition" seems like a more suitable label than "system" for the interacting collection of information technology components.

A number of strategies offer complementary approaches for achieving robustness in this setting, as suggested by Figure 1.

There are two general ways to deal with the possibility of bad things happening: *Prevent* them from happening at all and detect problems and *react* to them as they occur. We approach the former through validation and the latter through remediation. Within each of these categories we can identify (at least) three interesting cases.

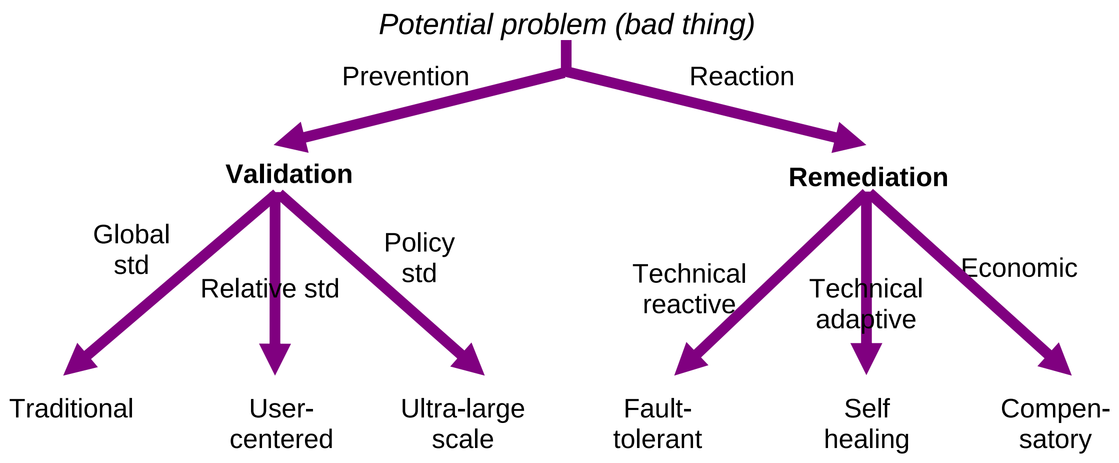

Figure 1: Approaches to robustness in modern software coalitions

Prevention

Prevention based on a global standard

This case is the focus of much of formal language theory and static program analysis, which attempt to make guarantees about programs based on the code of a system. In recent years this has focused on specific properties rather than complete specifications. The objective here is absolute guarantees.

This is also the focus of dynamic analysis including testing and dynamic analyses (e.g., of runtime behavior).

Prevention based on a relative standard

It is increasingly clear that the acceptability of a system to a specific user – and hence the robustness of the system in the eyes of that user – depends as much on the expectations of the user as it does on compliance with system specifications. Two issues arise.

First, the expectations of a given user may be either less demanding or more demanding than the system specification promises (or would promise if it existed). The first case is clear: the user might not use all of the precision, capability. or performance of the system or might be more tolerant of failures. The second case also arises, though: users often imagine what they hope the system may do and are unhappily surprised when it does not meet their expectations.

The second issue is a question of engineering cost-effectiveness: the cost of increasing robustness may not be justified by a user's actual needs. Rather, we need a way for individual users to determine whether a system is *sufficiently dependable* for their own needs.

Prevention based on a policy standard

We are beginning to come to grips with systems that are very large as measured by observable metrics such as lines of code, numbers of users, amount of data, and dependencies among components. But we continue to reason about them as if they were discrete systems subject to central control.

A more complex form of system is now emerging, with the Internet as a principal example. These systems are not simply larger versions of the systems we are familiar with. They feature

- Decentralized operation and control
- Conflicting, unknowable, diverse requirements
- Continuous evolution and deployment
- Heterogeneous, inconsistent, changing elements
- Indistinct people/system boundary
- Normal failures triggered by complex system coupling

These features preclude central design. These systems grow organically as a result of the actions of independent, possibly competitive users. They require new forms of acquisition and policy that are more akin to zoning laws – ways to govern independent evolution – than to conventional system specifications. [Software Engineering Institute. *Ultra Large-Scale Systems: The Software Challenge of the Future*. 2006 http://www.sei.cmu.edu/uls/]

Reaction

Reaction based on traditional reactive techniques

This case is the focus of classical fault tolerance, with roots in classical hardware fault tolerance with explicit set points or specifications of error states. In this case, robustness thresholds are set explicitly and crossing a threshold triggers remedial action. The strategy is to characterize the states of the system and the transitions between those states.

Reaction based on adaptive techniques

A difficulty with traditional reactive techniques is that they must invest specification effort in precisely defining the internal states or thresholds. Sometimes the robustness property of interest is appropriately treated as a threshold, but more often a system degrades gradually from dependable to undependable operation. Choosing an exact threshold requires making a choice of a single point in this gray area of decline.

An alternative is to base reaction on adaptive techniques that respond with low intensity to mild decline and with increasing intensity as the situation deteriorates, but that do so as a general reaction to conditions rather than as an explicit state change. Biological *homeostasis* offers tantalizing examples.

Robustness can also be improved by budgeting computing capability for reflection: maintaining a model of expected system behavior, monitoring system performance, and triggering adaptation. In effect, this replaces classical fixed setpoints with a more sophisticated basis for adaptation.

Reaction based on economic mechanisms

Sometimes systems fail and dynamic recovery is not possible. The world at large recognizes this as a risk management problem. One common way to manage such risks is to convert low probability, high impact events into high probability, low impact events. Insurance is a common example: risks of low probability, high cost events are pooled over a population that shares similar risks. Each member of the pool contributes a “premium” – a know payment that creates a fund that is subsequently disbursed to the few members of the pool who actually encounter the event.

Insurance-based risk management is common in software-intensive businesses, but it has received little attention at the system level. Creating an insurance model for software-intensive systems would require the ability to predict failure rates for the insured system, a way to attribute system failures to specific components, a way to evaluate the cost of a failure, and a way to create the risk-sharing pool.

Explicit risk management also offers an opportunity to make triage decisions – to assess system degradation and drop nonessential functions. This approach is used in provisioning certain types of service bureaus: an economic decision may provide to maintain less capacity than potential peak load, planning to drop (and pay penalties) some clients when load peaks in order to provide capacity for higher-priority clients.