NORTH ATLANTIC TREATY ORGANISATION

RESEARCH AND TECHNOLOGY ORGANISATION

AC/323(IST-047)TP/45

www.rto.nato.int

RTO TECHNICAL REPORT **TR-IST-047**

Building Robust Systems with Fallible Construction

(Elaboration de systèmes informatiques robustes à l'architecture faillible)

Final Report of the Task Group IST-047/RTG-019.

Published April 2008

Distribution and Availability on Back Cover

NORTH ATLANTIC TREATY ORGANISATION

RESEARCH AND TECHNOLOGY ORGANISATION

AC/323(IST-047)TP/45

www.rto.nato.int

RTO TECHNICAL REPORT **TR-IST-047**

Building Robust Systems with Fallible Construction

(Elaboration de systèmes informatiques robustes à l'architecture faillible)

Final Report of the Task Group IST-047/RTG-019.

The Research and Technology Organisation (RTO) of NATO

The content of this publication has been reproduced
directly from material supplied by RTO or the authors.

Published April 2008

ISBN 978-92-837-0049-4

Table of Contents

Building Robust Systems with Fallible Construction

(RTO-TR-IST-047)

Executive Summary

This is the final report of the Task Group IST-047/RTG-019 on "Building Robust Systems with Fallible Construction".

The general area of study that the task group was to investigate is related to Software Fault Tolerance, a topic that has been studied at least since 1970. Worldwide much has been learned about how to address those problems, as they were understood at the time.

However changes in perspective as to what constitute the challenges, and changes in available and commonplace technology, have led to a need go beyond conclusions reached in the past.

Today's systems are typically integrated from components. These components may themselves contain flaws, originating in specification, design or implementation errors, or in miscommunication between different teams involved in the development. More seriously, the integration process itself may be flawed, as when pre-existing components are used for purposes their developers had not envisioned, and the integrators misunderstand the detailed behaviour of the components. Interoperability of systems is more complex than correctness of a single system by itself. We have come to recognize that systems-of-systems have emergent behaviour, because the constituent subsystems were not only never designed as part of an integrated whole, they may actually be procured, owned and operated by independent organizations and have operational demands for results not encompassed within, or even aligned to, the objectives of the super-system itself. Components, and especially subsystems, often have an evolutionary life cycle independent of the life cycle of any system they are incorporated in: what may have been true at some point in time is not guaranteed to remain true in the future.

The Task Group focused on identifying challenges that have not been adequately resolved by traditional Software Fault Tolerance. The Task Group did not have the resources to itself undertake research to produce solutions, but felt that producing a catalogue of issues requiring further investigation was a useful first step leading to their eventual resolution, and in itself was a worthwhile contribution.

Today's NATO military systems depend on large, complex software with the need to be built and deployed more rapidly and cheaply than traditional development methods can deliver. Moreover, because military commanders depend on these systems, they must be more predictable and trustworthy than traditional development methods can deliver for the available time and cost investments.

Elaboration de systèmes informatiques robustes à l’architecture faillible

(RTO-TR-IST-047)

Synthèse

Ceci est le compte-rendu final du groupe de travail IST-047/RTG-019 sur « Elaboration de systèmes informatiques robustes à l’architecture faillible ».

Le domaine général sur lequel le groupe de travail a travaillé concerne la tolérance logicielle aux pannes, sujet étudié depuis au moins 1970. Partout dans le monde, beaucoup a été fait à l’époque pour résoudre ces problèmes.

Toutefois, des changements de perspective sur les défis que cela représente et les changements technologiques courants disponibles, ont abouti à la nécessité d’aller au-delà des conclusions passées.

De nos jours, dès les composants, les systèmes sont intégrés. Ces composants peuvent en eux-mêmes avoir des défauts, qui trouvent leur source dans les caractéristiques, la conception, les erreurs de mise en œuvre, ou une mauvaise compréhension entre les équipes de développement. Plus sérieusement, l’intégration elle-même peut être entachée d’erreurs, comme lorsque des composants déjà existants servent à des fins non prévues par les développeurs, sans parler des intégrateurs qui comprennent mal le comportement spécifique de ces composants. L’interopérabilité des systèmes est plus complexe que l’exactitude d’un seul système en lui-même. Il nous faut bien reconnaître que des systèmes-de-systèmes ont des comportements surprenants, car leurs sous-systèmes n’ont pas été conçus comme partie intégrante d’un tout. Ils peuvent même être achetés, détenus ou exploités par des organisations indépendantes, et être soumis à des exigences d’exploitation non comprises dans ou même alignées sur les objectifs du super-système lui-même. Les composants, et particulièrement les sous-systèmes, ont un cycle de vie autonome, indépendant du cycle de vie du système dans lequel ils sont incorporés : il est nullement garanti que ce qui a pu être vrai à un moment donné dans le temps le sera à l’avenir.

Notre groupe de travail s’est concentré sur l’identification des défis qui n’ont pas été résolus correctement par la traditionnelle tolérance logicielle aux pannes. Notre groupe n’avait pas les ressources pour lui-même rechercher les solutions, mais il a pensé que produire une liste des problèmes nécessitant des investigations plus poussées était une étape utile conduisant à leur résolution éventuelle, et en lui-même une contribution qui en valait la peine.

De nos jours les systèmes militaires de l’OTAN dépendent de gros programmes complexes, qui doivent être élaborés et déployés plus rapidement, encore moins chers que les méthodes traditionnelles de développement n’en sont capables. En outre, comme des commandants militaires dépendent de ces systèmes, ces derniers doivent être plus prévisibles et fiables que les méthodes traditionnelles de développement ne le permettent à ce jour en fonctions des investissements.

Chapter 1 – BACKGROUND AND MOTIVATION

The Terms of Reference for IST-047/RTG-019 (Annex D) were extremely ambitious. Quoting from the Justification (Relevance for NATO):

> "Today's NATO military systems depend on large, complex software with the need to be built and deployed more rapidly and cheaper than traditional development methods can deliver. Moreover, because military commanders depend on these systems, they must be more predictable and trustworthy than traditional development methods can deliver for the available time and cost investments. However this requirement is not quite compatible with the traditional project oriented view of software development, which is prevalent in today's military acquisition methods.
>
> Today's systems are typically integrated from components. These components may themselves contain flaws, originating in specification, design or implementation errors, or in miscommunication between different teams involved in the development. More seriously, the integration process itself may be flawed, as when pre-existing components are used for purposes their developers had not envisioned, and the integrators misunderstand the detailed behaviour of the components. Interoperability failures between different national systems often are of this form."

The general area of study that the Task Group was to investigate is related to Software Fault Tolerance, a topic that has been studied at least since 1970, when the Dependable Software Group was formed at the University upon Newcastle-on-Tyne in the UK. Worldwide much has been learned about how to address those problems, as they were understood at the time.

Software fault tolerance initially was construed to address design faults in software [Randell 1997]. More specifically, it addressed software that had a single thread of control, where control of that single thread was not lost when a failure occurred (as might occur on a wild transfer, access to a bad address or execution of a tight infinite loop) and where the failure could be recognized immediately by some oracle or by voting. Many failures that can occur are outside this restricted model. Later the model was broadened to include any situation where software failed to deliver its intended service, a concept labeled dependability [Laprie 1985]. This includes acknowledgement that a correct but tardy result may not be a satisfactory service. One benefit of this broader perspective is that it encompassed not only faults of commission, where the software explicitly did something incorrectly, but also faults of omission, where the software simply failed to handle situations that could arise. For example, since Tymshare, GEIS, and other multi-site timesharing services of the 1960s, distributed systems have had to handle network partitioning failures [Melliar-Smith 1998]. The initial fault is not a software error but a communications outage, whereby a collection of distributed processes designed to communicate with each other suddenly are partitioned into two or more non-communicating clusters. The software must not only continue to operate in this condition, but when communications is restored, must accomplish the often more difficult challenge of re-integrating the separate computational activities. The software fault is to ignore the possibility of this situation and thus not have code to recognize and treat it.

Another benefit of the broader definition is that it encompasses not only failures due to design errors, but also errors caused by input data falling outside the domain of applicability of the particular algorithm used. For many computations, different algorithms with different domains of applicability exist, and, without actually trying the algorithms, it may not be possible to recognize which algorithm will work for a particular case, let alone which is best. Thus even without coding errors, a completely achievable computation may fail. Of course data input from external sources should not be trusted to be appropriate. Patterson asserts that

68% of all outages of service on the Internet arise not from software bugs, but from inappropriate input from humans, specifically systems operators.

Nevertheless, even with the broader perspective, many of the approaches to dependability that have been proposed continue to have limited applicability. For example, numerous authors recommend journalizing input to a service, so that if the service should fail, it can be rolled back and retried with the same data. Software is typically deterministic: unless non-determinism is introduced somehow, given the same initial state and trying a service request again using the same method with the same input will trigger exactly the same failure again!

An early approach proposed to treat design faults was called *recovery blocks*. A recovery block, a unit of program structure, enclosed each questionable computation. After the computation, an oracle was invoked to assess whether the result was acceptable. If so, the block exited, but if not the computational state reverts to that on entry to the block, and the computation was attempted again another way, once more subject to assessment by the oracle. Many alternative algorithms might exist, but if none succeeds, the whole block fails. One drawback to recovery blocks for real-time systems is that processing time is obviously unpredictable.

The perspective that failures are caused by design faults gave rise to the recognition that the problem is not strictly technological: people are part of the problem because software developers caused those faults. This led to another early and intuitively attractive approach, *n-version implementation*. The idea was that if several disjoint teams each implemented different versions of the same specification, the dependability of the combined suite should be better than any single version, because presumably the teams would inject independent faults. While this can help, Knight and Leveson [Knight 1986] showed the benefit is not as great as might be hoped, because in fact even groups implementing different specifications in different languages make the same kinds of errors. The cause of such common mode faults is not completely understood: are some things just innately hard for programmers to get right, or, for instance, is the phenomenon a consequence of the way programmers are taught? Today a new source of common mode faults exists: some software development tools are widespread, and failures in these tools can introduce similar faults into completely unrelated software produced by completely unrelated teams!

Yet another early approach was the idea of *self-checking implementations*. If the designers understand what might go wrong, and can identify how incipient failures could be detected and averted, then implementations could be more dependable. For instance, in our experience 75% or more of the code of typical embedded systems is concerned with recognizing and handling exceptional situations. There are two challenges for this approach. First, from observation programmers are notoriously poor at anticipating what might go wrong with their programs, perhaps again a consequence of the way programmers are trained. (Even some authors explicitly writing about exception handling have overlooked plausible ways in which their designs may potentially fail.) A second cause of “sunny day” software design and development is that budget and delivery date pressures often cause programmers and, perhaps more seriously, their managers to give short shrift to addressing what might go wrong, because it detracts from completing the primary functionality of the product.

An excellent summary of the state-of-the-art is [Pullum 2001]. The Task Group was aware of this work and saw no reason to repeat it. Instead, the Task Group was chartered to investigate questions beyond the usual formulation of the problem. The Task Group organized a research workshop, IST-064/RWS-011, to elicit the collective wisdom of a wider community of experts. That workshop is reported separately.

Chapter 2 – CHANGING CONTEXT: NEW PERSPECTIVES

Changes in perspective as to what constitute the challenges for software fault tolerance have led to a need go beyond conclusions reached in the past. New perspectives represent different ways in which we need to think about problems, and different aspects that a proposed solution must address. No new uniquely military perspectives were found; on the other hand each of the perspectives discussed below occurs for military software. What does distinguish military applications is that the relative importance of different perspectives can change rapidly under battlespace conditions, whereas in the commercial world, priorities are more static.

2.1 ROBUSTNESS NOT NECESSARILY CORRECTNESS

The name of RTG-047 was specifically chosen to use the term “Robust” in order to make the point that our concern is graceful degradation in the presence of failures. This in general might be a weaker criterion than correctness, but it also may encompass other aspects that “correctness” does not necessarily address, such as space or time resource consumption, induced delays in response, performance predictability, etc. Our literature search turned up very little in approaches for robustness beyond the software fault tolerance literature.

2.2 PEOPLE ARE PART OF THE SYSTEM

A good example of accepting people as part of the system is that whereas residual faults in software were initially considered a technical problem to be addressed by strictly technical means, eventually it was appreciated that in most situations people are also part of the system, and accommodating their foibles may involve more than simple adjustment of technology. The early and intuitively attractive approach of *n-version implementation* constituted the recognition that software development teams were the source of the errors, which led to the proposal that multiple teams independently implementing the same software might result in the combined product being more dependable than that of a single team.

Another good example of this has been the recognition of the importance of “user friendliness” of systems for end users, those who work directly with screens, keyboards, mice, trackballs and so forth, entering data and responding to displayed results. User interface design has focused largely on making such user interfaces more intuitive, easier to learn, faster to operate, more likely to be adopted than avoided, but most of all less error prone with mistakes easier to undo [Patterson 2003]. These advances are important because if the end user enters the wrong input, or takes the wrong action in response to the software produced results, then the combined human-computer system is not fault tolerant.

2.3 DEPENDABILITY REQUIREMENTS DEPEND ON WHICH STAKEHOLDER

Today’s software is larger and more complex, and the implementation, deployment, operation and maintenance involve people in more roles than just as software developers or end users. These stakeholders play a part in a system meeting its service objectives, and their foibles play a part in a system failing to meet its objectives, i.e. in system failure. The roles these stakeholders play are not just coders and integrators, they include system architects, system operators, network management, security monitors, regulators, decision makers: indeed all those who affect or are affected by the system. These stakeholders are not homogeneous and may have different organizational objectives. They have different levels of understanding of the system, different opportunities to damage the system, as well as different opportunities to work around problems and recover from damage inflicted by system incidents. Dependability is not an absolute requirement, but the level required is relative to the needs of each stakeholder. Setting goals and capability restrictions for each of the roles is only part of managing the impact these people can have on a system. Understanding psychological and social factors, i.e. seeing the system as a socio-technical system, is important too, and especially important is addressing education and training issues such as culture shift, change management and pedagogy in bringing new people into these roles or sustaining existing staff.

2.4 AUTOMATED CORRECTION OF FAILURES IS NOT ALWAYS FEASIBLE OR APPROPRIATE

One significant shift in objectives has been the realization that the original objective of software fault tolerance, i.e. through diversity to automate the improvement in dependability by correcting errors, is not always feasible or appropriate. Instead, sometimes system failures need to be regarded as inevitable, and attention instead needs to be focused on recognizing the occurrence of a failure and providing tools to facilitate support for human intervention in recovery of the damaged system to an operational state. The UC Berkley and Stanford University project in Recovery-Oriented Computing has taken the lead in initiating work in this area [Patterson 2002] [Patterson 2002A].

2.5 AUTONOMIC COMPUTING, I.E. SELF-MANAGED SYSTEMS, HAS A ROLE

Somewhat in contrast to this, there also has been a trend in recent years for systems to be self-configuring, self-tuning, self-adapting, self-healing, etc. Various names have been applied to this initiative, a prominent one being autonomic computing [Kephart 2003]. Autonomic systems still are differentiated from traditional software fault tolerance in that the autonomic elements (the monitoring of system performance that is used at run-time to drive the adjustment to the system) are done out-of-line, using sensors and monitors that are much more sophisticated than the simple oracles or voting procedures that typify traditional software fault tolerance. The reconfiguration or tuning procedures of autonomic systems are also more sophisticated than the simple acceptance-or-rejection used with traditional software fault tolerance.

2.6 ROLLBACK IS NOT ALWAYS FEASIBLE OR DESIRABLE

Increasingly it is being appreciated that rollback may not always be feasible nor may it even be a desirable approach to fault tolerance. Rollback to an earlier state is only acceptable if the failed computation had no external effect. It is obvious for an embedded system that undoing an action such as firing a weapon is not possible. Even mere display of information may initiate human activity that cannot be undone. Passage of time alone may render rollback to an earlier state impossible: the external world itself has moved on, and the saved internal state can no longer match an external state that no longer exists. Service-Oriented Architectures (SOA), where services are obtained from third party suppliers, can exhibit similar effects. *Forward Error Correction* is not simply rollback followed by replay rolling forward again with more careful reprocessing of journalized input. Nor is *Forward Error Correction* simply the application of compensatory transactions to undo deleterious effects of the failed processing. Instead *Forward Error Correction* involves identifying an appropriate one of possibly many safe states from which processing can continue, and making the transition to that state. Transactions and the all-or-none ACID properties (Atomicity, Consistency, Isolation, Durability) may not always be relevant. Real-time video imagery provides an instructive illustrative example: processing failures are only objectionable if the viewer notices them, but that viewer is accustomed to many acceptable distractions.

2.7 SERVICE AVAILABILITY MAY OUTWEIGH CORRECTNESS OF INDIVIDUAL SERVICE REQUESTS

Somewhat related to this is a perspective change associated with the popular client-server model. Instead of a computation being regarded as strictly sequential units of work, the server abstraction processes units of work on demand. Correct processing of any individual service request (unit of work) is typically of lesser importance to the service owner than that the server must return to a state where it can continue to accept further service requests. Consequently, some individual service requests may get dropped on the floor. Although service requests may arrive sequentially, there is no implication that they are necessarily processed in that sequence, nor even that that they are processed atomically. Processing of multiple service requests may overlap. They may not complete in the sequence that they were accepted. While this behaviour is perhaps easiest to think about if each service request is treated as a separate thread of control, the effect can occur even when the server executes with a single thread of control, simply because service requests may block and have to be queued while awaiting external events, such as disk I/O, requests to other servers, or user input – and these external events may occur in unpredictable order. If a single service request fails, it may be discarded or rolled back, but rolling back other service requests that were successfully processed concurrently is usually neither desirable nor necessary.

2.8 SOFTWARE DEVELOPMENT IS NOT A SINGLE HOMOGENEOUS ACTIVITY

Abandonment of simplistic assumptions of homogeneity characterizes several categories of new perspectives. One is that in the past the software development activity was seen as homogeneous, with the corresponding work product being homogeneous: software. For today's systems, many quite different activities are involved in moving a software system from concept to operation, including software architecture, design, implementation and packaging. Fielding software to multiple sites each with differing requirements introduces more activities with development aspects, including installation after perhaps several levels of configuration and tailoring, ongoing maintenance, evolution and upgrading. These different activities produce different work products and are susceptible to different kinds of faults, but also facilitate and are limited to different mechanisms for fault detection and correction.

2.9 THE SOFTWARE PRODUCT MAY NOT BE HOMOGENEOUS CODE

Another example of going beyond homogeneity already noted is that whereas traditionally the software produced was treated as homogeneous, today many software products are composed of components, and that failures can arise not only within these components but also from the integration activity that combines these components. We noted further that large software systems may not just be made up of subsystems, but may be systems-of-systems, where effects are accomplished by the interoperation of independently owned, procured, and operated systems that maintain their individual purposes and responsibilities: operating organizations are not always homogeneous [Meier 1999].

2.10 THE DEVELOPMENT ORGANIZATION MAY NOT BE A HOMOGENEOUS ENTITY

The development organization, too, was traditionally treated as a single homogeneous entity. However, components are often procured from different suppliers, especially when those components are *Commercial-*

Off-The-Shelf (COTS) products or are obtained from *Open Source* organizations. The lack of a single design authority, with power to enforce design decisions throughout a system, can be drastic enough in itself, but the fact that the particular system being integrated is only one customer in the marketplace for those components, means that those responsible for that system have a negligible role in specification of the functionality of each component, and have even less influence in the timing or direction of the evolution of the component. Yet system implementation cannot remain static: evolution to track changes in the environment is essential for long-lived systems. Reliance on detailed internal behaviour of a component cannot yield robustness. Among other things, this has led to a not very satisfactory attempt [OMG2000] to handle software fault tolerance through special middleware.

2.11 MALICIOUS ATTACKS MAY BE AN ESSENTIAL CONCERN

Experience with networks, especially the Internet, has unfortunately led us to another new perspective, software today must be resistant to malicious attacks. Again this is usually an issue of avoiding faults of omission: we must anticipate attacks that might be tried, and ensure that code is available to detect and defend against them, as well as compensating for damage such attacks might produce.

2.12 FAULT TOLERANCE AWARENESS NEEDS TO BE INGRAINED IN STAKEHOLDERS

The final new perspective that we believe is needed is educational, to ingrain fault sensitivity awareness into trainee programmers, but also into other stakeholders from project managers to procurement officers to system operators. Software today is widely held not to be dependable, but while there is need for better techniques with wider applicability, the most glaring oversight is that available techniques are not widely known, or not given sufficient priority to be used. Once software has been deployed to the field, improvements in dependability are unlikely.

Chapter 3 – CHANGING CONTEXT: IMPACT OF TECHNOLOGY

The impact of technology also has led to a need go beyond conclusions reached in the past. Technology can give new options as to how to address software fault tolerance. It can also create new challenges that must be addressed. No new uniquely military technology impacts were found; on the other hand each of the impacts of technology discussed below occurs in military software.

We have identified four technologies that have become widely available in the past few years, each of which offers potential for new solutions to building more robust systems.

3.1 SURFEIT OF COMPUTING CAPACITY

The first of these is the surfeit of computing capacity. Most of the history of computing has been dominated by the need for faster, more powerful computers to address ever larger and more difficult problems, as well as the complementary need to find ways for software to use these computers more effectively on such problems. The flip side of the same coin has been the need for software to accomplish needed computation on the smallest, least expensive computers practical for these computations. Today, however, for most purposes the situation has changed: we have far more computing power than we know what to do with. To some extent this is a consequence of the server perspective: processors configured for peak loads have unused capacity during normal operation. In a mass production world, processor power has more to do with acquisition volume than with individual load projections, which generally leads to overcapacity. Moreover, at the chip level excess capacity is even more directly a consequence of reaching the limits of Moore’s law: power and heat considerations create barriers to running the clock faster, so extra gates on the chip are used for multiple cores and hyper-threading. Since most computations are still programmed for single thread sequential execution, this available parallelism is underutilized. This creates an opportunity to apply that capacity for fault tolerance, although it might have to be off the critical path of the single thread of control. To date, most approaches to fault tolerance require use of that single thread because they are synchronous with the primary computation. The best-known exception to this is the use of audit routines, a technique used since the 1960s in telecommunications software for on-line asynchronous verification and repair of data structures [Willet 1982]. Audit routines are also commonly used in verification and repair of file system data structures and main memory storage pools and heap structures. A key aspect of audit routines is that they do not require an instantaneous snapshot of the data structures, but instead are designed to perform their validation and repair concurrently with the main computation allocating, deallocating and modifying the data structures.

3.2 AUTONOMIC COMPUTING

The second technology impact likely to offer new options for software fault tolerance we have already commented on: the rise of autonomic computing. Although this technology benefits from the previously discussed surfeit of computing power, the real driver for the technology has been the increasing dependence on 24/7 operation of large complex software systems which are implemented as commercial software products rather than as custom designs. (Commercial software products refer to those sold in essentially identical form to many different customers with differing individual requirements.) This means that individual customers of the commercial software product are faced with installation activities requiring configuration and tailoring, as well as with ongoing tuning and adapting in operation. Typically such systems are not isolated, but must interoperate with other systems, existing and future. This installation and support effort is often not simply algorithmic, but is highly skilled, based on experience and heuristics, including long-term observation of system behaviour. The goal of autonomic computing is to eliminate, reduce, automate or deskill this effort through monitoring of appropriate sensors and analyzing and acting on the observations with algorithms, heuristics, and artificial intelligence. The choice of trouble sensors to date has been very application specific, as have been the corrective measures. Some of this is in use today, but there are many unresolved questions. Particularly relevant for our interest is the dependability of adaptive systems, even if the managed system itself is dependable. Furthermore, empirical evidence from testing adaptive systems can be hard to get: the system's internal actions to correct what it sees as aberrant conditions can make it difficult to find external input for black box tests to exercise particular situations.

3.3 VIRTUAL MACHINES

The third technology with impact likely to create new options for software fault tolerance is virtual machines. This term is used in two senses, and both are relevant. The first sense is that many software systems, such as language processors from PostScript to Java, execute through an interpreter rather than directly in the native machine language. This interpreter constitutes a virtual machine that exposes the computational operations and data types more directly than they can be impugned from the runtime state of compiled native machine language programs. Software fault tolerance techniques applied to programs for this virtual machine may well make better choices of state comparisons, recovery points, etc. The other sense in which the term virtual machine is used is with respect to an encapsulation of the external environment of a running program. Products such as Microsoft's Virtual PC or VMWare's Virtual Workstation and Virtual Server isolate running software from its actual environment, facilitating interception of any interaction with the actual environment, stopping the virtual processor clock to inspect or modify the state of the virtual program, save and restore (possibly much later, and possibly on a different host) virtual program states, and support of (multiple) virtual machines on a single host machine with quite different configuration. Companies are already commercially offering fault tolerance solutions exploiting this technology. We expect to see many more.

3.4 THE DISCIPLINE OF SOFTWARE ARCHITECTURE

The fourth technology impact likely to offer new options for software fault tolerance is the advent of the discipline of software architecture. Software architecture design as an identifiable activity in the software development process, and software architecture description a distinct deliverable from that process, provide a higher-level abstraction of the structure and behaviour of a software system. This gives a means to represent and analyze approaches to fault tolerance that are not at the statement-by-statement level, nor even at a local program structure level. Software architecture, for instance, can succinctly represent how a distributed application is allocated across processors, as well as explicitly indicating replication and redundancy of data and computational processes. Service restoration actions can be identified explicitly. Concurrency, or even actual parallelism, can be readily addressed. Architectural styles (patterns) enable such solutions to be generic, not tied to the design of a specific system. Service Oriented Architectures, with alternate service suppliers for particular services, suggest possible solutions as well as posing new challenges.

We have also identified a dozen technologies that have become prominent in the past few years that present new challenges for building robust systems.

3.5 SOFTWARE COMPONENT BASED ENGINEERING

The first technology to present new challenges is software component based engineering. The concept of software components was first introduced at the initial workshop on software engineering [McIlroy 1969] as a fundamental change in software development: instead of spelling out each detailed operation and representation by writing programs, we assemble systems by integrating existing but probably not quite matching components. This vision took over 30 years to become a marketplace reality, but today it is widespread. In the meantime, our understanding of the potential and challenges of this approach has become much deeper [Szyperski 2002]. We have learned, for instance, that programming language features and practices that seemed appealing in the context of writing complete programs from scratch are less than good ideas in the context of component based construction. Nevertheless, with the exception of FT-CORBA and other attempts to use middleware to address fault tolerance (and those relate only to restricted classes of failures), there appears to be very little that has been done for software fault tolerance in the context of software components. Components themselves internally can be fault tolerant, in an attempt to be trustworthy, but little has been studied about what internal requirements should be imposed, and what needs to be passed in and out of the component, to best arrange for the integrated ensemble to be fault tolerant or at least robust. Little has been studied about how to conduct the integration process to minimize the possibility of introducing faults. This indeed was the primary objective set for the Task Group, but we had no resources to conduct investigations, and in any case unfortunately we failed even to come up with novel ideas to investigate. Known problems needing solutions include dealing with architectural mismatches when components come from different suppliers and coping with component evolution that occurs independent of the evolution of the integrated system. These occur both with use of COTS components and with use of components that are shared with other products in a common product line.

3.6 SYSTEMS OF SYSTEMS

The second technology to present new challenges is the trend of assembling systems of systems. The subsystems in systems-of-systems are software components, but systems-of-systems go beyond software components because even at run-time, the subsystems maintain their separate identity, possibly even with users other than the integrating system. The independent management of these subsystems, responding to their various responsibilities and obligations, may not be willing to accommodate software fault tolerance instructions from the integrating system. A request for a cold restart might be a short-term example; a request to defer installing an upgrade might be a longer-term example. In many situations, it may be more appropriate to regard the ensemble as a set of interoperating systems rather than as a single system-of-systems, but we know very little about how to provide software fault tolerance for interoperating systems.

Military units often are dependent on a multitude of operational systems, which frequently have been conceived of, procured, and are operated as independent silos. A major effort in recent years has been to eliminate wasteful repetitiveness and reduce inconsistency by encouraging these silos to interoperate. The scale of the problem causes some putative solutions to be counterproductive. One US Army division reported depending on over 200 separate systems, and had serious training issues with the fact that these silos each installed new releases on its own schedule. The effort and coordination to synchronize upgrade cycles of less than 10 of these was enormous – but in the end it was recognized that all that had been accomplished was to create a bigger and more inflexible silo [Barr 2001].

3.7 WEB AND INTERNET TECHNOLOGIES

Internet and Web technologies present the third set of new challenges. The Internet has introduced many new paradigms, from how applications are deployed by download at run-time from remote servers to the use of XML as a universal representation of arbitrary data structures and their content. Use of HTTP and HTML is widespread, but these standards have limitations and indeed deficiencies as the rise of AJAX demonstrates.

Proxy servers, mirroring and other forms of caching introduce replicates of data that rollback recovery schemes must ensure get refreshed. Multimedia, especially time-based streaming media with separate synchronized streams, poses non-standard issues for fault tolerance. Search-based computations, possibly incorporating text analysis or link analysis, depend on the instantaneous state of the universe of Internet subscribers, as do content-based applications. Client-server and multi-tier architectures are common, with lower level services possibly provided by third parties (which with object request brokers and with service discovery through SOAP, WSDL and UDDI may be unknowable and not reproducible). To meet performance demands, service requests are often performed out of order, with unrelated service requests intervening if a server becomes available when a particular request blocks during processing. Viruses and pro-active spyware can modify the intended execution history of software. These technologies may invalidate assumptions underlying approaches to software fault tolerance. At the very least, they complicate the implementation of many approaches. Moreover, it is not obvious how to recognize some of the failures that may occur or what to do about them.

3.8 CONCURRENT, PARALLEL, AND DISTRIBUTED COMPUTING

Concurrency, indeed actual parallelism, has become ubiquitous today, as has distributed computing, and this constitutes the fourth set of challenges. The novelty for fault tolerance is two-fold. First, new failure modes must be handled, with new criteria for what constitutes an acceptable resolution. Network partitioning failures, as described earlier, illustrate this. Another illustration is given by considering how to restart the multiple processes of a failed distributed application [Shapiro 2004]. Second, fault tolerance solutions that mimic what is appropriate for sequential computation may be unacceptable, because they impose lockstep synchronization on processes that prevent the very performance enhancement opportunities that led to the use of multiple processors in the first place.

A powerful abstraction that has been put forward to address concurrency challenges for software fault tolerance is Coordinated Atomic Actions (CA) [Randell 1995]. CA is a unified scheme for coordinating complex concurrent activities and supporting error recovery between multiple interacting components in a distributed object system. Conversations (enhanced with concurrent exception handling: processes must be able to throw exceptions in other processes) are used to control cooperative concurrency and to implement coordinated error recovery whilst transactions are used to maintain the consistency of shared resources in the presence of failures and competitive concurrency. CA have often been used to establish all-or-none semantics – which we have already noted may not always be desired. Over more than ten years of research, this abstraction has been shown to be applicable to a wide range of distributed and concurrent situations, including web service implementations. What does not appear to have been studied is the downside of this abstraction: what are its limitations and what are its disadvantages?

3.9 EXCEPTION HANDLING

Fault tolerance is all about how to handle situations that should not occur in normal processing. Treating such situations is called exception handling. The seminal paper on exception handling dates from the mid-70s [Goodenough 1975], although there has been considerable work since. Exception handling was not part of early programming languages, so support for it had to be provided by operating systems or run-time libraries. Today, however, many languages have standardized on a common abstraction for exception handling: the try, throw and catch syntax and semantics. Unfortunately, relative to what was described in the seminal work, this abstraction is deficient. The common abstraction appears to take the position that exceptions represent software failures, perhaps unexpected, and that the only response is to abandon part of the computation, perhaps popping multiple levels off the activation stack. (The implicit release of storage can impose unpredictable run-time overheads.) That is, they have chosen the termination model, and ignored the resumption model. This is indeed one, but only one, of the positions suggested in the seminal work. At least as important is the idea that the normal path of execution should for clarity and performance only contain the dominant sequence of computation, and that all others, which will be abnormal in the sense of less frequent but which are by no means erroneous, should be expressed as exceptions. Rescaling a computation only if necessary to avoid overflow might be a simple example; a document formatter only doing widow processing when the text flow reaches the end of page might be another. IEEE floating-point arithmetic was specifically designed for the resumption model. In these cases discarding partial results is normally not desirable, moreover, when an exception is thrown, we need operand and state information as to what caused the exception so that the computed results can be adjusted and computation continued. Exceptions that return parameters are recommended in the seminal work, but recent languages exception mechanisms have disregarded this altogether. Exception handling in the presence of multiple processes, multiple threads, or other concurrency is simply a mess. Consequently, despite current language provisions, it is often still necessary for exception handling to be supported by operating systems, run-time libraries, or inline code.

3.10 NON-IMPERATIVE PROGRAMMING

Most techniques for software fault tolerance assume an imperative programming model and depend on, indeed modify, explicit control flow. These techniques then do not accommodate software expressed in a non-imperative paradigm: declarative programming languages, implicit invocation, data-driven synchronization, and tuple spaces for example.

3.11 GENETIC AND MORE GENERALLY EXPLORATORY COMPUTATION

Genetic and exploratory computations are optimization techniques that search large candidate spaces in analogy to biological population processes. Genetic and exploratory programming apply this optimization approach to the space of programs for computing a particular result. Genetic and exploratory computations exhibit the opposite problem to non-imperative programming: the algorithms can be intrinsically self-healing, so the appropriate response to many errors is simply to let the algorithm take care of any error. There is some evidence that the algorithms produced by genetic or exploratory programming are more robust against failures of subcomputations than typical algorithms for the same problems manually produced by top down design. We could imagine formulating a desired computation as the objective of genetic or exploratory computation to take advantage of this robustness. Approaches to making software-programmed hardware most robust to failure have been demonstrated using field-programmable gate arrays chips that are genetically programmed and able to survive and/or recover from radiation-based faults [Lohn 2006].

3.12 MASSIVE DATASETS

We have already noted that web-based search computations can depend on datasets so vast, and in such a continuous state of flux, that rollback and retry are inconceivable. Datasets not shared on the web can exhibit this too. They can be so large that exhaustive examination in response to a query is infeasible, and computation is only practical by caching (memoizing) partial results of previous computations and by doing speculative partial computations. Building on these may not produce results corresponding to any consistent state of the full dataset, but may in a reasonable period of time produce results that are good enough. Data mining might be a typical application.

3.13 INADEQUACY OF ORACLES

A fundamental assumption of many approaches to software fault tolerance is that there is an oracle that immediately identifies incorrect results. Unfortunately, for many computations this assumption is unrealistic. If the result is a data structure more extensive than a simple scalar (e.g. a large matrix), evaluation of the oracle may be expensive and itself potentially error prone. Indeed, for some computations leading to an extensive result (e.g. a GIS system producing a map) it is not clear what would constitute a meaningful criterion of correctness (derived maps depend on the quality of the geospatial data in the layers being combined, and such data typically has questions as to its accuracy, resolution, timeliness and pedigree). Internal consistency checks may be used for validation, but don't guarantee the result as a whole. Obtaining alternative results for comparison or plausibility checks is sometimes possible, but this may involve delay which not only may raise questions as to how comparable these alternates really are, but casts the whole all-or-none semantics into question, as service must continue to be provided after the results first become available. Voting also is only straightforward for simple scalar results: how big a difference is too big, and is arithmetic differencing meaningful on complex data structures. The notion of autonomic elements applying multiple trouble sensors to evaluate and analyze plausibility seems much more reasonable. Furthermore, in many systems there is no single result produced in isolation, instead there is a continuously evolving result as the input conditions change. Such systems seem more amenable to closed loop management, where steering is applied manually or autonomously to reduce error signals at the trouble sensors.

3.14 SECURITY AND PRIVACY

Security and privacy have become big issues for many software systems today, and these are qualities for which it seems particularly unlikely that satisfactory oracles to identify failures can be found or that any failure detection will be immediate. Remedial action when failure has been detected is also not obvious: rollback and retry would seem singularly inappropriate.

3.15 MULTIMEDIA, ESPECIALLY TIME-BASED STREAMING MEDIA

Many modern software systems deal with multimedia, especially time-based streaming media. It is commonplace for several streams to need to be synchronized: video and sound, speech perhaps in each of several simultaneous translations, for example. Here too characterizing what constitutes a failure is not obvious, finding a suitable oracle to detect it is unlikely, and identifying suitable remediation beyond dropping frames and re-synching is inordinately challenging.

3.16 SCALABILITY AND NON-STOP OPERATION

Many critical systems today, especially military systems, must operate non-stop 24/7. It is not unusual that such systems must be scalable; both in that instances of such a system need to be deployed in situations with widely different demand, but also that it is not uncommon for demand to grow over time, perhaps even to grow rapidly. Fault tolerance and robustness is particularly important for such systems, yet the non-stop operation and the scale and the need to grow make it particularly hard to apply current failure detection and remediation techniques.

3.17 RATE OF NEW RELEASES

The final technology to present a new challenge for robustness is simply the turmoil of continuous updates from different suppliers. We earlier alluded to the distress this caused a US Army unit with respect to training. The Web is never in a consistent or reproducible state, which makes interoperability a nightmare, but nevertheless crucial. We normally try to engineer systems for a predictable static state, but this is more like riding a living, writhing monster. We need new approaches to avoid it bucking us off!

Chapter 4 – CONCLUSIONS AND FOLLOW-ON

IST-047/RTG-019 did not achieve its objective of discovering solutions to the challenges of building robust systems with construction known to be fallible. Nevertheless, we feel that we have performed a useful service in scoping out the challenges, especially in identifying new perspectives from which putative solutions will be viewed, and current technologies that may support new approaches but certainly represent a richer class of problems to address. The next step for researchers and vendors worldwide will be to fit these challenges into a research agenda to tackle the challenges, and come up with products and services that take them into account.

Chapter 5 – REFERENCES

[Barr 2001] Barr, B. "Role of Test and Evaluation in Evolutionary Procurement" in NATO NC3A Symposium Evolutionary Procurement of Information Systems, EPIS 2000, the Hague, 28-30 March 2001.

[Cristian 1991] Cristian, F. "Understanding Fault-Tolerant Distributed Systems", CACM Vol. 34, No. 2, 1991.

[Goodenough 1975] Goodenough, J.B. "Exception handling: Issues and a proposed notation", CACM Vol. 18, No. 12, 1975, pp. 683-696.

[Kephart 2003] Kephart, J.O. and Chess, D.M. "The Vision of Autonomic Computing", IEEE Computer, January 2003, pp. 41-50.

[Knight 1986] Knight, J.C. and Leveson, N.G. "An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming," IEEE Transactions on Software Engineering, Vol. SE-12, No. 1, January 1986, pp. 96-109.

[Laprie 1985] Laprie, J.C. "Dependable Computing and Fault Tolerance: Concepts and terminology" in Proc. 15th IEEE Int. Symposium on Fault-Tolerant Computing (FTCS-15), Ann Arbor, Michigan, 1985, pp. 2-11.

[Lohn 2006] Lohn, J.D. and Larchev, G. "Evolutionary Based Techniques for Fault Tolerant Field Programmable Gate Arrays," Proc. 2006 Space Mission Challenges for Information Technology, June 2006.

[Meier 1999] Meier, M.W. "Architecting Principles for Systems-of-Systems", Systems Engineering, 2:1, 1999.

[McIlroy 1969] McIlroy, M.D. "Mass produced Software Components", in P. Naur and B. Randell, *Software Engineering, Report on a Conference Sponsored by the NATO Science Committee, Garmisch Germany 7th to 11th October, 1968*, Scientific Affairs Division, NATO Brussels, 1969, pp. 138-155.

[Melliar-Smith 1998] Melliar-Smith, P.M. and Moser, L.E. "Surviving Network Partitioning", IEEE Computer Vol. 31, No. 3, 1998, pp. 62-68.

[OMG 2000] Object Management Group, *Fault Tolerant Corba: Modifications to Corba Core Specifications*, 2000 http://www.omg.org/docs/ptc/00-03-05.pdf.

[Patterson 2002] Patterson, D.A., Brown, A., Broadwell, P., Candea, G., Chen, M., Cutler, J., Enriquez, P., Fox, A., Kiciman, E., Merzbacher, M., Oppenheimer, D., Sastry, N., Tetzlaff, W., Traupman, J. and Treuhaft, N. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. UC Berkeley Computer Science Technical Report UCB//CSD-02-1175, March 15, 2002.

[Patterson 2002A] Patterson, D.A. "Recovery Oriented Computing: A New Research Agenda for a New Century", HPCA 8 keynote, 2002, www.cs.berkley.edu/~patterson/talks/keynote.html.

[Patterson 2003] Brown, A. and Patterson, D.A. "Undo for Operators: Building an Undoable E-mail Store". In Proceedings of the 2003 USENIX Annual Technical Conference, San Antonio, TX, June 2003 (Best Paper Award).

[Pullum 2001] Pullum, L.L. *Software Fault Tolerance Techniques and Implementation*, Artech House Publishers, 2001.

[Randell 1995] Randell, B., Romanovsky, A., Rubira, C., Stroud, R., Wu, Z. and Xu, J. “From Recovery Blocks to Coordinated Atomic Actions” In *Predictably Dependable Computer Systems* Eds. B. Randell, J.-C. Laprie et al., Springer-Verlag, 1995, pp. 87-101.

[Randell 1997] Randell, B. “Dependability Research at Newcastle”, 1997, http://www.cs.ncl.ac.uk/events/anniversaries/40th/webbook/dependability/index.html.

[Shapiro 2004] Shapiro, M.W. “Self-Healing in Modern Operating Systems”, ACM Queue, Vol. 2, No. 9, December 2004, pp. 67-75.

[Szyperski 2002] Szyperski, C. *Component Software: Beyond Object-Oriented Programming (2nd Edition)*, Addison-Wesley Professional, 2002.

[Willet 1982] Willet, R.J. “Notes from Design of Recovery Strategies for a Fault-Tolerant No. 4 ESS”, BSTJ Vol. 61, No. 10, 1982.

Annex A – REVIEW OF HISTORY OF TASK GROUP

The proposal to initiate an Exploratory Team on this topic arose in spring 2002 during a discussion at the IST Panel 9th Panel Business Meeting. The initial draft TAP and ToR were submitted for consideration at the 22/23 September 2002 at the IST Panel 10th PBM, but it appears no ET was formally approved. Instead, the Task Group IST-047/RTG-019 was approved October 2003 for immediate start as follow-on to an earlier Task Group.

The successful kick-off meeting was held at the RTA in Paris, March 11 and 12, 2004. The Tap and ToR were revised, and a Program of Work agreed to. That was the last physical meeting of the whole Task Group. An informal meeting of a subset of the Task Group took place 1 October 2004 in den Haag. Attempted meetings in Rome and Athens in 2005 and Orlando in spring 2006 failed to take place because of lack of travel funds. Nevertheless, the Task Group continued its activities by electronic mail.

The Task Group organized the successful IST-064/RWS-011 Workshop November 2006 in Prague. This workshop is reported separately.

Annex B – TASK GROUP MEMBERS

CANADA (CHAIRMAN)

Dr. W. Morven GENTLEMAN	Tel.	+1 902 494 2652
Global Information Networking Institute	Fax	+1 902 492 1517
Dalhousie University	E/M:	Morven.Gentleman@dal.ca
6050 University Avenue		
Halifax, Nova Scotia B3H 1W5		

CZECH REPUBLIC

Dr. Milan SNAJDER	Tel.	+420 (2) 55 70 87 53
Military Technical Institute of Air Force	Fax	+420 (2) 55 70 84 53
VTULaPVO	E/M	milan.snajder@vtul.cz
Mladoboleslavska 944		
19721 Prague 97		

NETHERLANDS

Mr. Yves Van de VIJVER	Tel.	+31 (20) 511 36 71
National Aerospace Laboratory	Fax	+31 (20) 511 32 10
Anthony Fokkerweg 2	E/M	vyver@nlr.nl
PO Box 90502		
1006 BM Amsterdam		

UNITED KINGDOM

Dr. Peter POPOV	Tel.	+44 (0) 20 7040 8963
Center for Software Reliability	Fax	+44 (0) 20 7040 8585
City University	E/M	ptp@csr.city.ac.uk
Northampton Square		
London EC1V 0HB		

UNITED STATES

Dr. Jason LOHN	Tel.	+1 (650) 604-5138
MS 269-1	Fax	+1 (650) 604-3594
NASA Ames Research Center	E/M	jlohn@arc.nasa.gov
Mountain View, CA		

Annex C – PROPOSED RTO TECHNICAL PROGRAMME AND BUDGET 2004

ACTIVITY	Task Group	**BUILDING ROBUST SYSTEMS WITH FALLIBLE CONSTRUCTION**											03/2003		
Activity REF. Number	IST-047												01/2004		
PRINCIPAL MILITARY REQUIREMENTS		1	2	5	6					NU			12/2006		
MILITARY FUNCTIONS		1	2	3	4	7	9								
PANEL AND COORDINATION		IST (Information Systems Technology)								NC3A					
LOCATION AND DATES		Participating Nations, Semi-annually											P-I		
PUBLICATION DATA		TR (Final Report)						03/2007			200		NU		
KEYWORDS	Alternate	System Integration					Components						Redundancy		
Development Methodology		Plausibility Oracle					Data Fusion								

C.1 BACKGROUND AND JUSTIFICATION (Relevance to NATO)

Today’s NATO military systems depend on large, complex software with the need to be built and deployed more rapidly and cheaper than traditional development methods can deliver. Moreover, because military commanders depend on these systems, they must be more predictable and trustworthy than traditional development methods can deliver for the available time and cost investments. However this requirement is not quite compatible with the traditional project oriented view of software development, which is prevalent in today’s military acquisition methods.

Today’s systems are typically integrated from components. These components may themselves contain flaws, originating in specification, design or implementation errors, or in miscommunication between different teams involved in the development. More seriously, the integration process itself may be flawed, as when pre-existing components are used for purposes their developers had not envisioned, and the integrators misunderstand the detailed behaviour of the components. Interoperability failures between different national systems often are of this form.

Experience with interoperating commercial products, especially in the context of the Internet, indicates that robustness to fallible components and fallible integration can be achieved without centralized predictive coordination. Appropriate software architecture, redundancy in functional components, and enforcement of critical interface standards appear to be key elements of success. Improved registry and plug-and-play concepts can help automate integration and reduce configuration problems.

If we are to try to build infallible systems with fallible construction methods, there is a need to review the advances in software development in the commercial market. There is also a need to evaluate the requirements of military software development, bring forth lessons to be learned and to identify areas of research and draw projections especially for the procurement community.

C.2 OBJECTIVE(S)

Although the commercial community has been very active in such new methods we do not see any considerable projection of that activity on the military side, where it is most needed. The commercial marketplace has long been dealing with independently developed interoperable products, where it has been recognized that each product will be upgraded across many releases over its lifetime, in order to meet changing requirements and to take advantage of new technologies.

C.3 TOPICS TO BE COVERED

- Choices for software architecture for robustness
- Integration process and tools
- Critical interface standards
- Interoperability with complementary or related products
- Empirical behaviour investigation through testing
- Oracles to ascertain plausibility of results
- Coping with component evolution
- Aids to retraining users
- Scaffolding reuse
- Regression tests, integration tests, integrity testing, consistency testing
- Project metric tracking
- Implications for cultural change

C.4 DELIVERABLES AND/OR END PRODUCT

Technical Report (TR): Final Report at the end of the mandate of the Task Group.

C.5 TECHNICAL TEAM LEADER AND LEAD NATION

Recommended Team Leader:	Morven GENTLEMAN, CAN.
Recommended Lead Nation:	CAN.

C.6 NATIONS WILLING TO PARTICIPATE

CAN, GBR, NLD and USA.

C.7 NATIONAL AND/OR NATO RESOURCES NEEDED (Physical and Non-Physical Assets)

Members of the Task Group will be experts in software development and acquisition.

Host Nations will provide meeting arrangements. No special needs are foreseen except for Internet access.

C.8 RTA RESOURCES NEEDED (e.g. Consultant Funding)

Support could be asked if needed for one of two Consultants per year.

Annex D – TERMS OF REFERENCE

NATO AC/323 Information Systems Technology Panel
Task Group on Building Robust Systems with Fallible Construction
(IST-047/RTG-019)

D.1 ORIGIN

A) Background

The idea for an Exploratory Team on Infallible Systems from Fallible Components was first proposed in the ninth IST Panel meeting in 2002. From consultation with interested parties, it became clear that the integration process itself is at least as fallible as the components. This was the Achilles' heel of earlier investigations, such as recovery blocks in the 1970's Moreover, setting the objective as high as "infallibility" could lead to concentration on techniques impractical in practice, whereas a more modest goal might be entirely satisfactory and achievable. This ToR takes these comments into account.

B) Justification (Relevance for NATO)

Today's NATO military systems depend on large, complex software with the need to be built and deployed more rapidly and cheaper than traditional development methods can deliver. Moreover, because military commanders depend on these systems, they must be more predictable and trustworthy than traditional development methods can deliver for the available time and cost investments. However this requirement is not quite compatible with the traditional project oriented view of software development, which is prevalent in today's military acquisition methods.

Today's systems are typically integrated from components. These components may themselves contain flaws, originating in specification, design or implementation errors, or in miscommunication between different teams involved in the development. More seriously, the integration process itself may be flawed, as when pre-existing components are used for purposes their developers had not envisioned, and the integrators misunderstand the detailed behaviour of the components. Interoperability failures between different national systems often are of this form.

Experience with interoperating commercial products, especially in the context of the Internet, indicates that robustness to fallible components and fallible integration can be achieved without centralized predictive coordination. Appropriate software architecture, redundancy in functional alternatives, and enforcement of critical interface standards appear to be key elements of success. Improved registry and plug-and-play concepts can help automate integration and reduce configuration problems.

If we are to try to build infallible systems with fallible construction methods, there is a need to review the advances in software development in the commercial market. There is also a need to evaluate the requirements of military software development, bring forth lessons to be learned and to identify areas of research and draw projections especially for the procurement community.

D.2 OBJECTIVES

1) Area of Research and Scope

Although the commercial community has been very active in such new methods we do not see any considerable projection of that activity on the military side, where it is most needed. The commercial marketplace has long been dealing with independently developed interoperable products, where it has been recognized that each product will be upgraded across many releases over its lifetime, in order to meet changing requirements and to take advantage of new technologies.

2) The Specific Goals and Topics to be Covered by the Task Group are:

- Choices for software architecture for robustness;
- Integration process and tools;
- Critical interface standards;
- Interoperability with complementary or related products;
- Empirical behaviour investigation through testing;
- Oracles to ascertain plausibility of results;
- Coping with component evolution;
- Aids to retraining users;
- Regression tests, integration tests, integrity testing, consistency testing;
- Scaffolding reuse;
- Project metric tracking; and
- Implications for cultural change.

3) Expected End Products and/or Deliverables

Final Report.

4) Overall Duration of the Task Group

The Technical Group lifetime is proposed for three years.

D.3 RESOURCES

A) Membership

The Task Group is expected to deliberate on the specific topics given above, at IST Panel meetings and over the Internet, in 1 – 2 day meetings, twice a year.

Recommended Team Leader: Dr. Morven GENTLEMAN (CAN). Recommended Lead Nation: Canada. (A final Leader will be elected during the first meeting of the Task Group).

Initial list of Nations that have expressed a willingness to participate: CAN, GBR, NLD and USA.

B) National and/or NATO Resources Needed

Each participating nation is expected to provide at least .2 person-years/year of effort towards the goals of this RTG, plus funds to allow their experts to travel to two RTG meetings per year. Members of the Task Group will be experts in software development and acquisition.

In addition, Internet access is required for unclassified information exchange and collaborative activities. Identified RTA resources would be limited to standard support for publishing the final report. RTA support for consultants may also be requested. Support could be asked if needed for one or two Consultants per year.

D.4 SECURITY CLASSIFICATION LEVEL

Recommended for the activity: NATO Unclassified and commercially confidential.

Recommended for the publication: NATO Unclassified.

D.5 PARTICIPATION BY PARTNER NATIONS

Partner Nations are invited to participate in the Technical Team.

D.6 LIAISONS

Liaison with NATO C3A would enhance the work of the Task Group.

REPORT DOCUMENTATION PAGE

1. Recipient's Reference	2. Originator's References	3. Further Reference	4. Security Classification of Document
	RTO-TR-IST-047 AC/323(IST-047)TP/45	ISBN 978-92-837-0049-4	UNCLASSIFIED/ UNLIMITED

5. Originator	Research and Technology Organisation North Atlantic Treaty Organisation BP 25, F-92201 Neuilly-sur-Seine Cedex, France
6. Title	Building Robust Systems with Fallible Construction
7. Presented at/Sponsored by	Final Report of the Task Group IST-047/RTG-019.

8. Author(s)/Editor(s)	9. Date
Multiple	April 2008

10. Author's/Editor's Address	11. Pages
Multiple	36

12. Distribution Statement	There are no restrictions on the distribution of this document. Information about the availability of this and other RTO unclassified publications is given on the back cover.

13. Keywords/Descriptors

Computer architecture	Integrated systems
Computer programs	Reliability
Design	Software development
Fault tolerance	System of systems

14. Abstract

The Task Group focused on identifying challenges that have not been adequately resolved by traditional Software Fault Tolerance. The Task Group did not have the resources to itself undertake research to produce solutions, but felt that producing a catalogue of issues requiring further investigation was a useful first step leading to their eventual resolution, and in itself was a worthwhile contribution.

DIFFUSION DES PUBLICATIONS
RTO NON CLASSIFIEES

Les publications de l'AGARD et de la RTO peuvent parfois être obtenues auprès des centres nationaux de distribution indiqués ci-dessous. Si vous souhaitez recevoir toutes les publications de la RTO, ou simplement celles qui concernent certains Panels, vous pouvez demander d'être inclus soit à titre personnel, soit au nom de votre organisation, sur la liste d'envoi.

Les publications de la RTO et de l'AGARD sont également en vente auprès des agences de vente indiquées ci-dessous.

Les demandes de documents RTO ou AGARD doivent comporter la dénomination « RTO » ou « AGARD » selon le cas, suivi du numéro de série. Des informations analogues, telles que le titre est la date de publication sont souhaitables.

Si vous souhaitez recevoir une notification électronique de la disponibilité des rapports de la RTO au fur et à mesure de leur publication, vous pouvez consulter notre site Web (www.rto.nato.int) et vous abonner à ce service.

CENTRES DE DIFFUSION NATIONAUX

ALLEMAGNE
Streitkräfteamt / Abteilung III
Fachinformationszentrum der Bundeswehr (FIZBw)
Gorch-Fock-Straße 7, D-53229 Bonn

BELGIQUE
Royal High Institute for Defence – KHID/IRSD/RHID
Management of Scientific & Technological Research for Defence, National RTO Coordinator
Royal Military Academy – Campus Renaissance
Renaissancelaan 30, 1000 Bruxelles

CANADA
DSIGRD2 – Bibliothécaire des ressources du savoir
R et D pour la défense Canada
Ministère de la Défense nationale
305, rue Rideau, 9e étage
Ottawa, Ontario K1A 0K2

DANEMARK
Danish Acquisition and Logistics Organization (DALO)
Lautrupbjerg 1-5, 2750 Ballerup

ESPAGNE
SDG TECEN / DGAM
C/ Arturo Soria 289
Madrid 28033

ETATS-UNIS
NASA Center for AeroSpace Information (CASI)
7115 Standard Drive
Hanover, MD 21076-1320

FRANCE
O.N.E.R.A. (ISP)
29, Avenue de la Division Leclerc
BP 72, 92322 Châtillon Cedex

GRECE (Correspondant)
Defence Industry & Research General Directorate, Research Directorate
Fakinos Base Camp, S.T.G. 1020
Holargos, Athens

HONGRIE
Department for Scientific Analysis
Institute of Military Technology
Ministry of Defence
P O Box 26
H-1525 Budapest

ISLANDE
Director of Aviation
c/o Flugrad
Reykjavik

ITALIE
General Secretariat of Defence and National Armaments Directorate
5th Department – Technological Research
Via XX Settembre 123
00187 Roma

LUXEMBOURG
Voir Belgique

NORVEGE
Norwegian Defence Research Establishment
Attn: Biblioteket
P.O. Box 25
NO-2007 Kjeller

PAYS-BAS
Royal Netherlands Military Academy Library
P.O. Box 90.002
4800 PA Breda

POLOGNE
Centralny Ośrodek Naukowej Informacji Wojskowej
Al. Jerozolimskie 97
00-909 Warszawa

PORTUGAL
Estado Maior da Força Aérea
SDFA – Centro de Documentação
Alfragide
P-2720 Amadora

REPUBLIQUE TCHEQUE
LOM PRAHA s. p.
o. z. VTÚLaPVO
Mladoboleslavská 944
PO Box 18
197 21 Praha 9

ROUMANIE
Romanian National Distribution Centre
Armaments Department
9-11, Drumul Taberei Street
Sector 6
061353, Bucharest

ROYAUME-UNI
Dstl Knowledge Services
Information Centre
Building 247
Dstl Porton Down
Salisbury
Wiltshire SP4 0JQ

SLOVENIE
Ministry of Defence
Central Registry for EU and NATO
Vojkova 55
1000 Ljubljana

TURQUIE
Milli Savunma Bakanlığı (MSB)
ARGE ve Teknoloji Dairesi Başkanlığı
06650 Bakanliklar
Ankara

AGENCES DE VENTE

NASA Center for AeroSpace Information (CASI)
7115 Standard Drive
Hanover, MD 21076-1320
ETATS-UNIS

The British Library Document Supply Centre
Boston Spa, Wetherby
West Yorkshire LS23 7BQ
ROYAUME-UNI

Canada Institute for Scientific and Technical Information (CISTI)
National Research Council Acquisitions
Montreal Road, Building M-55
Ottawa K1A 0S2, CANADA

Les demandes de documents RTO ou AGARD doivent comporter la dénomination « RTO » ou « AGARD » selon le cas, suivie du numéro de série (par exemple AGARD-AG-315). Des informations analogues, telles que le titre et la date de publication sont souhaitables. Des références bibliographiques complètes ainsi que des résumés des publications RTO et AGARD figurent dans les journaux suivants :

Scientific and Technical Aerospace Reports (STAR)
STAR peut être consulté en ligne au localisateur de ressources uniformes (URL) suivant: http://www.sti.nasa.gov/Pubs/star/Star.html
STAR est édité par CASI dans le cadre du programme NASA d'information scientifique et technique (STI)
STI Program Office, MS 157A
NASA Langley Research Center
Hampton, Virginia 23681-0001
ETATS-UNIS

Government Reports Announcements & Index (GRA&I)
publié par le National Technical Information Service
Springfield
Virginia 2216
ETATS-UNIS
(accessible également en mode interactif dans la base de données bibliographiques en ligne du NTIS, et sur CD-ROM)

NORTH ATLANTIC TREATY ORGANISATION

RESEARCH AND TECHNOLOGY ORGANISATION

BP 25

F-92201 NEUILLY-SUR-SEINE CEDEX • FRANCE

Télécopie 0(1)55.61.22.99 • E-mail mailbox@rta.nato.int

DISTRIBUTION OF UNCLASSIFIED RTO PUBLICATIONS

AGARD & RTO publications are sometimes available from the National Distribution Centres listed below. If you wish to receive all RTO reports, or just those relating to one or more specific RTO Panels, they may be willing to include you (or your Organisation) in their distribution.

RTO and AGARD reports may also be purchased from the Sales Agencies listed below.

Requests for RTO or AGARD documents should include the word 'RTO' or 'AGARD', as appropriate, followed by the serial number. Collateral information such as title and publication date is desirable.

If you wish to receive electronic notification of RTO reports as they are published, please visit our website (www.rto.nato.int) from where you can register for this service.

NATIONAL DISTRIBUTION CENTRES

BELGIUM
Royal High Institute for Defence – KHID/IRSD/RHID
Management of Scientific & Technological Research for Defence, National RTO Coordinator
Royal Military Academy – Campus Renaissance
Renaissancelaan 30
1000 Brussels

CANADA
DRDKIM2 – Knowledge Resources Librarian
Defence R&D Canada
Department of National Defence
305 Rideau Street, 9th Floor
Ottawa, Ontario K1A 0K2

CZECH REPUBLIC
LOM PRAHA s. p.
o. z. VTÚLaPVO
Mladoboleslavská 944
PO Box 18
197 21 Praha 9

DENMARK
Danish Acquisition and Logistics Organization (DALO)
Lautrupbjerg 1-5
2750 Ballerup

FRANCE
O.N.E.R.A. (ISP)
29, Avenue de la Division Leclerc
BP 72, 92322 Châtillon Cedex

GERMANY
Streitkräfteamt / Abteilung III
Fachinformationszentrum der Bundeswehr (FIZBw)
Gorch-Fock-Straße 7
D-53229 Bonn

GREECE (Point of Contact)
Defence Industry & Research General Directorate
Research Directorate, Fakinos Base Camp
S.T.G. 1020
Holargos, Athens

HUNGARY
Department for Scientific Analysis
Institute of Military Technology
Ministry of Defence
P O Box 26
H-1525 Budapest

ICELAND
Director of Aviation
c/o Flugrad, Reykjavik

ITALY
General Secretariat of Defence and National Armaments Directorate
5th Department – Technological Research
Via XX Settembre 123
00187 Roma

LUXEMBOURG
See Belgium

NETHERLANDS
Royal Netherlands Military Academy Library
P.O. Box 90.002
4800 PA Breda

NORWAY
Norwegian Defence Research Establishment
Attn: Biblioteket
P.O. Box 25
NO-2007 Kjeller

POLAND
Centralny Ośrodek Naukowej Informacji Wojskowej
Al. Jerozolimskie 97
00-909 Warszawa

PORTUGAL
Estado Maior da Força Aérea
SDFA – Centro de Documentação
Alfragide
P-2720 Amadora

ROMANIA
Romanian National Distribution Centre
Armaments Department
9-11, Drumul Taberei Street
Sector 6, 061353, Bucharest

SLOVENIA
Ministry of Defence
Central Registry for EU and NATO
Vojkova 55
1000 Ljubljana

SPAIN
SDG TECEN / DGAM
C/ Arturo Soria 289
Madrid 28033

TURKEY
Milli Savunma Bakanlığı (MSB)
ARGE ve Teknoloji Dairesi Başkanlığı
06650 Bakanliklar – Ankara

UNITED KINGDOM
Dstl Knowledge Services
Information Centre
Building 247
Dstl Porton Down
Salisbury, Wiltshire SP4 0JQ

UNITED STATES
NASA Center for AeroSpace Information (CASI)
7115 Standard Drive
Hanover, MD 21076-1320

SALES AGENCIES

NASA Center for AeroSpace Information (CASI)
7115 Standard Drive
Hanover, MD 21076-1320
UNITED STATES

The British Library Document Supply Centre
Boston Spa, Wetherby
West Yorkshire LS23 7BQ
UNITED KINGDOM

Canada Institute for Scientific and Technical Information (CISTI)
National Research Council Acquisitions
Montreal Road, Building M-55
Ottawa K1A 0S2, CANADA

Requests for RTO or AGARD documents should include the word 'RTO' or 'AGARD', as appropriate, followed by the serial number (for example AGARD-AG-315). Collateral information such as title and publication date is desirable. Full bibliographical references and abstracts of RTO and AGARD publications are given in the following journals:

Scientific and Technical Aerospace Reports (STAR)
STAR is available on-line at the following uniform resource locator: http://www.sti.nasa.gov/Pubs/star/Star.html
STAR is published by CASI for the NASA Scientific and Technical Information (STI) Program
STI Program Office, MS 157A
NASA Langley Research Center
Hampton, Virginia 23681-0001
UNITED STATES

Government Reports Announcements & Index (GRA&I)
published by the National Technical Information Service
Springfield
Virginia 2216
UNITED STATES
(also available online in the NTIS Bibliographic Database or on CD-ROM)

ISBN 978-92-837-0049-4