Local Reconfiguration Policies [*]

Jonathan K. Millen
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
millen@csl.sri.com

Abstract

Survivable systems are modelled abstractly as collections of services supported by any of a set of configurations of components. Reconfiguration to restore services as a result of component failure is viewed as a kind of "flow" analogous to information flow. We apply Meadows' theorem on datset aggregates to characterize the maximum safe flow policy. For reconfiguration, safety means that services are preserved and that that reconfiguration rules may be stated and applied locally, with respect to just the failed components.

1. Introduction

System survivability is concerned with the ability (of a distributed computer system) to continue to make resources available, despite adverse circumstances including hardware malfunctions, software flaws, malicious user activities, and environmental hazards such as electronic interference ([9], p. 97). It is a higher-level property that includes computer and network security, fault tolerance, and assurance [10]. Survivability can be investigated from many points of view; our purpose is to apply an abstract model motivated by, and deriving specific results from, another model used in computer security.

A distributed system is characterized here as a collection of components configured to provide a set of user services. Different fault-tolerant architectures have different characteristics with regard to how many components may safely fail, whether components are interchangeable or whether certain components must be operational, and what kind of failure is acceptable. Sometimes failed components are assumed to be simply unresponsive, but some systems can deal with active misbehavior such as Byzantine faults, in which failed components act in an arbitrary and possibly malicious fashion.

In the present context we focus attention on those sets of components that are sufficient to support system services, when other components have failed. At this level of abstraction we are not concerned with the type of failure, and we will assume that it is known which components have failed. We address the question of determining how to reconfigure a system using replacement components in such a way as to restore the required services. This setting distinguishes our approach from work in the artificial intelligence arena that uses a similar abstract system model, but focuses on the problem of fault diagnosis for a single service, using incomplete knowledge of the system and its fault, given as a set of logical formulas. A typical paper in this category is [3]; and a paper that, unusually, takes this approach but does address reconfiguration is [1].

We call a set of components sufficient to support a service a *composition* of the service. The same service might be supported with different compositions. If components are interchangeable, perhaps any subset of a given size m of the total set is sufficient. For example, systems that use "voting" will work if any majority of an active set is operating correctly. Byzantine faults can often be overcome when at least two thirds of the components are correct. (See, for example, [6].)

Components might be interchangeable, yet for structural reasons there might be limits on which unused components can be reconfigured to replace failed ones, as in [8].

If components are not interchangeable, there might be distinct subclasses of components and some members of each subclass must be operational, as in systems like Pluribus [5]. In any case, given a system service, and knowing the fault-tolerance mechanism, we can determine the collection of all its supporting compositions.

The notion of "service" is not defined here except in the context of the compositions that support it and their configurations. The user or designer of a system is free to decide what services the system is expected to provide. Two

[*]This research was supported by the U. S. Army Research Laboratory under contract DAKF11-97-C-0020.

services supported by the same sets of components can be distinguished only if the components must be configured differently with respect to their ability to support other services simultaneously.

The service provided by a composition can depend on how the components in it are connected or which interfaces are used. As a trivial example, consider the capital letters I and V as components of a Roman numeral. Their composition is just the set $\{I, V\}$, but this composition can be configured either as IV or VI to support the incompatible services of signifying the number 4 or 6.

If the interconnection cannot be changed quickly and automatically, the two services are not available at the same time. Thus, in general, the availability of components in a composition is necessary but not sufficient to ensure availability of any particular service supported by the composition.

Before proceeding with the formalization, it may be useful to summarize what kinds of results could be achieved in the context of this simple model of services. Three types of results seem likely.

Representation. The composition model of a service is very general, but it would be inefficient to describe a service by actually listing the sets of components that support it. We can look for optimal, irredundant representations for service. The notion of a service "basis" that appears below is a step in this direction.

Algorithms. Given a collection of services in a suitable representation, algorithms could be derived to enumerate their dependencies, their shared structure, their sensitivity to failures, and so on. These algorithms, or the tasks themselves, can be analyzed for their computational complexity.

Insights. Taking an unconventional view of a problem may yield unique insights that would be unlikely to have been recognized in other formulations. An example of this is the "local reconfiguration" result in this paper.

Local reconfiguration is a property of a policy for transforming the current state of a system in response to failures. At a given time, certain components are allocated and configured to support a set of services. When some of those components fail, various options may exist for restoring the required services by replacing or bypassing the failed components. A reconfiguration policy is based on a set of (old, new) pairs of system configurations. Such a policy is *local* if the permissibility of an (old, new) pair can be answered simply by looking for a (replaced, replacement) pair, where the replaced set may, for example, consist of just the failed components. Locality means that there is a much more efficient representation of the policy than a list of system-level (old, new) pairs.

The local reconfiguration result is that, given a system and a specification of which services are supported by which components, there exists a unique maximal local reconfiguration policy. Surprisingly, this result follows from a result in multilevel computer security, due to Meadows [7], regarding the existence of a safe information flow policy in an environment (typically a database) where an aggregate of information from different objects may have an abnormally raised sensitivity.

2. Formal Model for Services

A key observation about services is the following: *if a given composition is sufficient to provide a service, then any larger composition will also be sufficient.* That is, the availability of more correctly operating components is never a disadvantage. One could argue that this is sometimes not the case, as for example if the additional components increase the load on a system and consequently decrease its performance below an acceptable level. But for a large class of systems, having more correctly-operating components is a benefit, or can be made to be a benefit by designing the components properly. Also, any disadvantage derived from having extra components can be neutralized by configuring the service with those components disconnected, so that they are treated as non-operational.

To identify compositions that support a service, we can just list the minimal compositions of the service, since any other composition supports the service if and only if it includes some minimal composition. A set of minimal compositions is *irredundant* in the sense that no two compositions in it are comparable via set inclusion.

Formally, a system expresses a mapping from a set of services to irredundant sets of compositions.

Definition 2.1 (System) *A pair $S = (\overline{S}, \underline{S})$ consisting of a set of* services $\overline{S}$ *and a set of* components $\underline{S}$ *is a* system *if there is a* basis *mapping $s \mapsto [s]$ defined on $\overline{S}$ such that for all $s \in \overline{S}$,*

1. $u \in [s] \Rightarrow u \subseteq \underline{S}$ *and*

2. $[s]$ *is irredundant:* $u, v \in [s]$ *and* $u \subseteq v$ *implies* $u = v$.

We refer to $[s]$ as the *basis* of s. In this paper we will deal with only one system S, so all references to services and components refer to elements of the given S.

It is reasonable to assume also that each element of a basis is finite. It is unnecessary to make the stronger assumption that there are only a finite number of components, or that each basis is finite, and it seems too heavy-handed to do so, given the ever-expanding size of the global Internet.

A composition supports a service if and only if it includes at least one of the basis compositions for that service. In this situation we will also say that the composition supports the basis.

Definition 2.2 (Support) *A composition u* supports *a service s if there exists $v \in [s]$ such that $v \subseteq u$.*

It follows immediately that basis elements are minimal with respect to set inclusion among compositions that support a service.

Not every irredundant set of finite compositions is necessarily the basis for some service in a given system. Many such sets would be associated with services that are nonsensical or simply unwanted. Consider the two special services whose bases are $\emptyset$ and $\{\emptyset\}$, respectively. The basis $\emptyset$ might be said to belong to an "impossible" service since no set of components is able to support it. The basis $\{\emptyset\}$ might be said to belong to a "don't-care" service since it is supported even when no components are available.

2.1. Survivability Pre-ordering of Bases

Services can be compared with respect to survivability. One service is at least as survivable as another if every composition that supports the other system will also support this one.

Definition 2.3 (Survivability Pre-ordering) *$s \sqsubseteq t$ if (u supports s implies u supports t.)*

The ordering $s \sqsubseteq t$ means that s is no more survivable than t. For, since any composition supporting s must support t, s cannot operate with any composition that fails to support t.

Note that the impossible service is the least survivable, and the don't-care service is the most survivable.

The relation $\sqsubseteq$ is only a pre-ordering on services, because although it is obviously reflexive and transitive, antisymmetry fails. When s and t are supported by the same compositions, they are not necessarily the same service, because, as remarked earlier, a composition may be configured and interconnected differently to provide different services.

The survivability of services can be compared by looking at their bases.

Proposition 2.1 (Basis Comparison) *$s \sqsubseteq t$ iff for all $u \in [s]$ there exists $v \in [t]$ such that $v \subseteq u$.*

Proof. Suppose $s \sqsubseteq t$ and let $u \in [s]$. Then u supports s, so u also supports t and there exists $v \in [t]$ such that $v \subseteq u$.

Conversely, suppose $u \in [s]$ implies there exists $v \in [t]$ such that $v \subseteq u$. Suppose w supports s. Then there exists $u \in [s]$ such that $u \subseteq w$. Produce $v \in [t]$ such that $v \subseteq u$. Then $v \subseteq w$, so w supports t. ∎

Corollary 2.2 (Basis Partial Ordering) *The relation $\sqsubseteq$ induces a partial ordering on bases. In particular, if $s \sqsubseteq t$ and $t \sqsubseteq s$ then $[s] = [t]$.*

Proof. This is easily checked. Antisymmetry follows from the fact that bases are irredundant. ∎

3. Safe Reconfigurations

Vint Cerf has said that "networks are always broken," meaning that in a large enough network, there are always some nodes that are malfunctioning or down, so that reconfiguration is a constant fact of life. A similar comment would apply to other fault-tolerant systems. This leads us to characterize the state of a system, in part, by the current set of operational components.

We may be interested in states that provide less than the entire set of services $\overline{S}$. That is because not all of the services offered by a system are necessarily needed at a given time. This is particularly true if the system in question is merely an internal subsystem of some larger, fault-tolerant system such as a network. A system may also operate in degraded modes in which some useful subset of the entire set of services is being maintained, because not enough components are available to support them all. Hence, a characterization of a system state should identify the set of services that it is configured to provide, and indicate which components are allocated to which services.

Definition 3.1 (States) *A state p of a system S is a pair $(\overline{p}, \underline{p})$ such that*

1. *$\overline{p} \subseteq \overline{S}$ is a set of services;*
2. *$\underline{p} \subseteq \underline{S}$ is a set of components called the* support *of p, such that $\underline{p}$ supports every $s \in \overline{p}$;*

Furthermore, there exists at least one function f on $\overline{p}$ called a configuration *of p such that*

1. *$f(s) \subseteq \underline{p}$ and*
2. *$f(s)$ supports s.*

A configuration is disjoint *if its range elements are pairwise disjoint.*

A configuration of a state p is a function that shows how each service in $\overline{p}$ is supported by sets of components in $\underline{p}$.

An important point here is that some configurations are *realizable* and some are not. Suppose, for example, that a service s is supported by a single component A, and that another service t is also supported by A. Can both s and t be supported in a state p with $\overline{p} = \{s, t\}$ and $\underline{p} = \{A\}$? The question is whether A can be shared to support both s and t at the same time, or whether it can only be configured to support one or the other. In the first case, p has a realizable configuration, and in the second case, it does not. A state with a realizable configuration is itself called realizable.

Realizability may depend on which other components are present. Certain kinds of peripheral storage devices, for example, may be attached to any of several workstations. They can be shared by two workstations concurrently if and only if there is a network connection between the two workstations.

Whether a state is realizable or not is an empirical fact about the sharability of components in it, so we do not know in advance which of the possible states are realizable. It depends on the system. The set of realizable states must be part of the characterization of a system.

Definition 3.2 (Realizable States) *Let the set of realizable states of the system S be denoted by $\mathcal{R}$. Every realizable state has a realizable configuration.*

It is reasonable to assume that deleting services or adding components (without connecting them, for example) does not destroy the realizability of a state. This is expressed as an axiom.

Axiom 3.1 (Spare) *If $p \in \mathcal{R}$, $d \subseteq \overline{p}$ and $\underline{p} \subseteq u \subseteq \underline{S}$, then $(d, u) \in \mathcal{R}$.*

It is reasonable to assume that any disjoint configuration is realizable, since no components have to be shared.

Axiom 3.2 (Disjoint Realizability) *If f is a disjoint configuration of p then f is realizable and $p \in \mathcal{R}$.*

3.1. Reconfiguration

A state transition is just a pair of realizable states $p \rightsquigarrow q$ representing a possible reconfiguration of the system. A failure in one or more components can force a transition.

It is desirable to maintain enough redundancy or repair capability so that when a failure causes the system to depart from one state p, it can be brought soon into another state q that supports at least the same services. Those transitions are called service-preserving.

Definition 3.3 (Service-Preserving) *A transition $p \rightsquigarrow q$ is* service-preserving *if $\overline{p} \subseteq \overline{q}$.*

It is tempting to try to localize the specification of a transition policy by saying something like, "if this component fails, replace it by this one," without considering the realizability constraints and the possible future impact fully.

As an example, suppose that some service s is supported normally by component A and t is supported normally by component B. Assume that there is a standby C that can replace either A or B but cannot be shared to do both. A transition policy to use C for either A or B has the defect that if the reconfiguration happens for A first, and then B fails, there will be no replacement for B. We might want to know this so that we can add another replacement component to the standby stock, or replace C with a sharable unit.

This sort of analysis can be facilitated by a result that arises from an unexpected quarter, namely, database information flow security. We will describe the original source first, then show how to apply it to this reconfiguration problem.

3.2. Dataset Aggregate Systems

Consider the problem of aggregation in multilevel secure databases. A set of data items has a sensitivity level that cannot decrease as the set is enlarged. When two sets of data items are merged, the combined sensitivity level may stay at the maximum of the combined levels or rise to a higher level. The "aggregation problem" as such refers to situations where the level of the merged set is strictly higher than the least upper bound of the levels of the sets merged. It has been discussed in papers such as [4].

In 1990, Meadows proved a theorem about information flow in aggregates, which we will refer to as the "Dataset Aggregate Theorem" [7]. With a suitable mapping of dataset concepts to service concepts, we can apply that theorem to obtain a nontrivial result about fault tolerance.

A dataset aggregate system is a triple (X, L, λ) where X is a collection of "datasets," L is a lattice of sensitivity levels, and $\lambda : 2^X \to L$ is a monotone function assigning a lattice element (ordinarily a sensitivity level) to each set of datasets. (2^X is the powerset of X, the set of its subsets.) The rationale for choosing a lattice as the domain of sensitivity levels appeared first in [2]. Monotonicity means that if $u \subseteq v$ then $\lambda(u) \leq \lambda(v)$. [1]

Meadows' result concerns (information) flow policies. A flow policy is a transitive relation R, written in infix form as $\to_R$, that extends set inclusion on sets of datasets. That is,

Definition 3.4 (Flow Policy) *$\to_R$ is a flow policy on X if*

transitive: *$\to_R$ is a transitive relation on 2^X, and*

monotone: *if $u \subseteq v \subseteq X$ then $u \to_R v$.*

A flow policy is *safe* if it respects the partial ordering of sensitivity levels, and aggregating source sets is permitted. That is,

Definition 3.5 (Safe Flow Policy) *A flow policy $\to_R$ on X is* safe *with respect to λ if*

λ-preserving: *$u \to_R v$ implies $\lambda(u) \leq \lambda(v)$ and*

[1] It will be helpful later to note that monotonicity implies that $\lambda(u) \vee \lambda(v) \leq \lambda(u \cup v)$, where '$\vee$' is the lattice join.

aggregative: *$u \rightarrow_R w$ and $v \rightarrow_R w$ implies $(u \cup v) \rightarrow_R w$.*

The idea behind safe flow policies is that if $u \rightarrow_R v$, then information can be permitted to flow from u to v, since a subject with sufficient access to view v would also have sufficient access to view u. If both u and v can flow to w, then their combined information can be aggregated in w, so w must have a high enough level to cover the aggregate level. The intent is to make sure that when flows are permitted, the destination object is labelled sufficiently high to protect against future aggregation.

Meadows' Dataset Aggregate Theorem states that there exists a unique maximal safe flow policy, and gives a characterization of it. This is Theorem 2.5 of [7].

Theorem 3.1 (Dataset Aggregate Theorem) *Let (X, L, λ) be a dataset aggregate system, and let $\rightarrow_R$ be the flow policy defined by $u \rightarrow_R v$ iff $\forall w, \lambda(u \cup w) \leq \lambda(v \cup w)$. Then $\rightarrow_R$ is the unique maximal safe flow policy.*

An important consequence of this theorem is that the maximal safe flow policy is determined by flows from single elements. This is part (a) of Lemma 2.6 of [7].

Lemma 3.2 (Element Flow) *Let $\rightarrow_R$ be the maximal safe flow policy on (X, L, λ). Then $u \rightarrow_R v$ if and only if, for each element $x \in u, \{x\} \rightarrow_R v$.*

3.3. Application to Survivability

The first thing we need to do is to construct a dataset aggregate system out of a system of services and components. In our application, datasets will be components, so that λ is defined on compositions. Our analogue of a sensitivity level is based on the idea that the more services a composition supports, the more sensitive or critical it is.

However, a given composition does not determine a single maximal set of services, since it may support different sets of services in different incompatible configurations. The following definition of λ seems to work. The next subsection explains why other, simpler, definitions fail.

Definition 3.6 (Composition Sensitivity Level) $\lambda_S(u) = \{\overline{p} | p \in \mathcal{R}$ *and* $\underline{p} = u\}$.

Thus, $\lambda_S(u)$, the sensitivity level of a composition u, is the set of service-sets of realizable states supported by u.

Let $D = 2^{\overline{S}}$ be the collection of sets of services. The set of sensitivity levels for λ_S is the powerset 2^D; any powerset is a lattice under set inclusion. Furthermore, λ_S is monotone with respect to set inclusion.

Proposition 3.3 (λ_S Monotone) *If $u \subseteq v$ then $\lambda_S(u) \subseteq \lambda_S(v)$.*

Proof. Suppose $u \subseteq v$. Let $\overline{p} \in \lambda_S(u)$. Then $\underline{p} = u \subseteq v$. By the Spare Axiom, $q = (\overline{p}, v) \in \mathcal{R}$, and $\underline{q} = v$, so $\overline{q} \in \lambda_S(v)$. But $\overline{q} = \overline{p}$, so $\overline{p} \in \lambda_S(v)$. ∎

With this choice for λ, we have a dataset aggregate system.

Proposition 3.4 (Aggregate System) $(\underline{S}, 2^D, \lambda_S)$ *is a dataset aggregate system.*

Recall that the first property of a safe flow policy is that safe transitions maintain λ. That is, if $u \rightarrow_R v$ is safe, then $\lambda(u) \leq \lambda(v)$. If this property is satisfied it has an interesting consequence for the existence of service-preserving transitions.

Theorem 3.5 (Service-Preserving Flows) *These two properties are equivalent:*

P1 $\lambda_S(u) \subseteq \lambda_S(v)$

P2 *for all $p \in \mathcal{R}$ such that $\underline{p} = u$ there exists $q \in \mathcal{R}$ such that $\underline{q} = v$ and $\overline{p} = \overline{q}$.*

Proof. Assume P1. Let $p \in \mathcal{R}$ such that $\underline{p} = u$. Then $\overline{p} \in \lambda_S(u)$. By P1, $\overline{p} \in \lambda_S(v)$. Hence there exists $q \in \mathcal{R}$ with $\underline{q} = v$ and $\overline{p} = \overline{q}$.

Assume P2. Let $\overline{p} \in \lambda_S(u)$. Then $\underline{p} = u$. By P2, there exists $q \in \mathcal{R}$ with $\underline{q} = v$ and $\overline{p} \subseteq \overline{q}$. By the Spare Axiom, $r = (\overline{p}, v) \in \mathcal{R}$ and we have both $\overline{r} \in \lambda_S(v)$ and $\overline{p} = \overline{r}$. Thus $\overline{p} \in \lambda_S(v)$. ∎

The property P1 is the first of the two properties of a safe flow relation. The property P2 says that any state supported by u can be reconfigured to a state supported by v with a service-preserving transition.

3.4. Why Other λs Fail

Other potential and apparently simpler choices for λ in a survivability context are either not monotone or do not yield the result on service-preserving flows.

Suppose, for example, we defined $\lambda_S(u) = \{s | u$ supports $s\}$. This sensitivity level is monotone, but it is not the case that P1 implies P2. If $\lambda_S(u) \subseteq \lambda_S(v)$, and p is realizable with $\underline{p} = u$, we know that each service in $\underline{p}$ is supported by v. However, v might only be able to support them individually, not all together, so there is no guarantee of a realizable reconfigured state q providing all of $\overline{p}$.

Another tempting choice is $\lambda_S(u) = \{p \in \mathcal{R} | \underline{p} = u\}$. This one is not monotone. We can make a monotone version with a small change: let $\lambda_S(u) = \{p \in \mathcal{R} | \underline{p} \subseteq u\}$. In fact, P1 implies P2 with this definition. But P2 does not imply P1. The problem is that this definition is too restrictive– one can show that if $\lambda_S(u) \subseteq \lambda_S(v)$ then $u \subseteq v$. This is too restrictive because one of the reasons we might want to replace u is that all its elements have failed, in which case we want v to be disjoint from u.

3.5. The Induced Reconfiguration Policy

We can apply this result to reconfigurations by defining a reconfiguration policy as a transition relation on realizable states. A flow policy induces a reconfiguration policy as follows.

Definition 3.7 (Induced Reconfiguration) *If $\rightarrow_R$ is a flow policy with respect to λ_S, the* induced reconfiguration policy *$\rightsquigarrow_R$ is defined by: $p \rightsquigarrow_R q$ if $p, q \in \mathcal{R}$ and $\underline{p} \rightarrow_R \underline{q}$.*

From definitions and the service-preserving flow result, it is immediate that a safe flow policy induces a service-preserving reconfiguration policy, and it is complete in the sense that any permitted flow induces a permitted reconfiguration.

Corollary 3.6 (Service-Preserving Reconfiguration) *Suppose that $\rightarrow_R$ is a safe flow policy. Then:*

1. *any reconfiguration $p \rightsquigarrow_R q$ is service-preserving, and*
2. *if $\underline{p} \rightarrow_R v$ then there exists q such that $\underline{q} = v$ and $p \rightsquigarrow_R q$.*

This corollary applies to any safe flow policy, but of course the greatest benefit is obtained by applying it to the maximal safe flow policy.

Definition 3.8 (Maximal Safe Reconfiguration) *If $\rightarrow_S$ is the maximal safe flow policy, $\rightsquigarrow_S$ is the* maximal safe reconfiguration policy.

The correspondence between the aggregation concepts and the reconfiguration concepts is summarized in the table below.

Aggregation	Reconfiguration
Datasets X	Components $\underline{S}$
Aggregates $u \subseteq X$	Compositions $u \subseteq \underline{S}$
Sensitivity level λ	$\lambda_S(u) = \{\overline{p} \mid p \in \mathcal{R}$ and $\underline{p} = u\}$
Flow policy $\rightarrow_S$	Induced reconfiguration policy $\rightsquigarrow_S$

3.6. Localization

Our next task is to look at the second property of a safe flow policy, the ability to aggregate source sets.

Suppose the transitions $u \rightarrow w$ and $v \rightarrow w$ are both permitted by a safe flow policy. Then $u \cup v \rightarrow w$ is also permitted. Thus, by the corollary above, for any state p with $\underline{p} = u \cup v$, there is a reconfiguration q with $\underline{q} = w$.

Here is a simple example of an unsafe flow policy. Assume that we have a system with components A, B, and C and services s, supported by A, and t, supported by B. Component C is a floating backup that can replace either A or B but not both at the same time. A policy permitting $\{A\} \rightarrow \{C\}$ and $\{B\} \rightarrow \{C\}$ is unsafe, because while each transition by itself preserves whatever service is supported, any attempt to replace the union $\{A, B\} \rightarrow \{C\}$ will fail.

The facts of the situation do not offer any alternatives, since there is no way to recover from failures of both A and B. The point is that we would like to know in advance whether this is the case, and checking safety will tell us.

The maximal safe flow policy satisfies the Element Flow lemma, which allows us to test for safe flows one element at a time. It has the following consequence for the induced reconfiguration policy.

Theorem 3.7 (Local Reconfiguration) *Let $\rightsquigarrow_S$ be induced by the maximal safe flow policy $\rightarrow_S$. Let $p \in \mathcal{R}$. Then these two conditions are equivalent:*

1. *there exists $q \in \mathcal{R}$ such that $\underline{q} = v$ and $p \rightsquigarrow_S q$*
2. *for all $x \in \underline{p}$ we have $\{x\} \rightarrow_S v$.*

The Local Reconfiguration Theorem simplifies the job of the system administrator who has to reconfigure the system after failures of components $x_1, x_2, \ldots$. It shows that all that is needed is a list giving, for each component, one or more replacement compositions. After choosing u_1 such that $\{x_1\} \rightarrow_S u_1$, and u_2, ... similarly, the theorem guarantees that there is a service-preserving reconfiguration to a state q such that $\underline{q} = \bigcup_i u_i$.

The list of possible replacements u for a component x need only include minimal sets, by transitivity of the maximal safe flow policy. Thus, the maximal safe reconfiguration policy can be represented by listing the minimal compositions u such that $x \notin u$ (since a failed component cannot belong to its replacement set) and $\{x\} \rightarrow_S u$.

4. Constructing the Maximal Safe Flow Policy

The Meadows paper addressed the computational problem of determining in practice, for a manageably small set of datasets, which flows between aggregates are permitted by the maximal safe flow policy. Her approach was to use the mapping g taking each aggregate to the set of all datasets that are permitted to flow into it. Meadows did not give a name for this mapping; we will call it the *flow closure* function.

Definition 4.1 (Flow Closure) *The* flow closure *g is defined by:*

$$g(u) = \{x \mid \{x\} \rightarrow_S u\}$$

Suppose $\rightarrow_S$ is the maximal safe flow policy. Because of the Element Flow lemma,

$$u \rightarrow_S v \iff u \subseteq g(v) \iff g(u) \subseteq g(v).$$

The Meadows paper did not explain in a general way how to generate the closures $g(u)$, although an example was given. A way to construct the flow closure for small systems is suggested here. We begin with a theorem that serves as the first step in identifying elements of $g(u)$. This result is an easy consequence of the Dataset Aggregate Theorem.

Theorem 4.1 (Element Closure) *$\{x\} \to_S u$ if and only if $x \in u$ or both $\lambda(\{x\}) \leq \lambda(u)$ and for all v such that $x \in v$, $\lambda(v) \leq \lambda(u \cup (v - \{x\}))$.*

The test in the Element Closure theorem is made for all v such that $x \in v$, but actually it is sufficient to make the test for the relatively small collection of *excepted* aggregates, defined by Meadows as follows.

Definition 4.2 (Excepted Aggregate) *A nonempty aggregate $u \subseteq X$ is* excepted *if*

$$\lambda(u) \neq \bigvee_{w \in \mathcal{P}'(u)} \lambda(w)$$

where $\mathcal{P}'(u) = 2^u - \{u\}$ is the set of proper subsets of u.

Excepted aggregates are the anomalous sets exhibiting the "aggregation problem." Note that, in our application, every set in a basis is excepted, because any proper subset would not support the service whose basis contains that set. This makes λ_S of the subset strictly smaller than that of the excepted aggregate.

Note that if u is excepted, then $\lambda(u) \neq \bigvee_{w \in W} \lambda(w)$ for any subcollection $W \subseteq \mathcal{P}'(u)$.

Theorem 4.2 (Excepted Aggregate Testing) *Suppose there exists v such that $x \in v$ and $\lambda(v) \not\leq \lambda(u \cup (v - \{x\}))$. Then any v that is minimal with this property is excepted.*

Proof. Let v be minimal with the property in the theorem. Let W be the set of proper subsets of v containing x. For each $w \in W$, $\lambda(w) \leq \lambda(u \cup (w - \{x\}))$. Hence

$$\bigvee_{w \in W} \lambda(w) \leq \lambda(u \cup (v - \{x\}))$$

since λ is monotone and $v = \bigcup W$. If v were not excepted, we would have $\lambda(v) = \bigvee_{w \in W} \lambda(w)$, contradicting the assumed property of v. ∎

Putting the last two theorems together, we have:

Theorem 4.3 (Testing) *$\{x\} \to_S u$ if and only if $x \in u$ or both $\lambda(\{x\}) \leq \lambda(u)$ and for all excepted aggregates v such that $x \in v$, $\lambda(v) \leq \lambda(u \cup (v - \{x\}))$.*

Thus, for a given aggregate u, we can find its flow closure in a time proportional to these factors: the size of X, the time it takes to compute λ, and the number of excepted aggregates. The number of aggregates is itself exponential in the size of X, so any application of this approach to large examples will have to take advantage of special properties of those examples, such as interchangeability of components of the same type.

We can simplify the job of identifying which aggregates are excepted by observing that it is sufficent to look at subsets just one element smaller.

Proposition 4.4 (Excepted Aggregate Identification) *If $u \neq \emptyset$, u is excepted if and only if $\lambda(u) \neq \bigvee_{x \in u} \lambda(u - \{x\})$.*

Proof. This condition is necessary since each $u - \{x\}$ is a proper subset of u. Sufficiency follows from the fact that for any proper subset $w \subseteq u$ there exists x such that $w \subseteq u - \{x\}$. ∎

5. Examples

There are a few situations in which it is easy to discern whether $u \to_S v$ in the maximal flow policy. For example, if there are no excepted aggregates, then $\{x\} \to_S u$ if and only if $\lambda(\{x\}) \leq \lambda(u)$, and consequently $u \to_S v$ if and only if $\lambda(u) \leq \lambda(v)$.

At the other extreme, suppose that the set of all datasets (or components, in our interpretation) is excepted. This means that there is some configuration in which every component is needed. In such a system, we can take the set of all components for v in the Testing result and conclude that $g(u) = u$ for all u. In other words, no component can be replaced by a set u unless it belongs to u, so it cannot be replaced if it has failed. This is an indication of how conservative the maximal flow policy can be. However, there are plenty of systems in which there are spare components, and safe reconfigurations are possible.

Consider a system in which every aggregate of size n is excepted, and no others. This is the case for λ_S if there is one service and it is supported by any n components. Smaller sets do not support the service, and larger sets do not support any further services. Then, if $x \notin u$, the testing result implies that $\{x\} \to_S u$ if and only if u has at least n elements (otherwise an n-element excepted set containing u would violate the inequality).

Here is a somewhat more elaborate example. In this system there are three types of replaceable components: processors, memory cards, and disk drives. Other components, such as monitors and printers, are outside the scope of the analysis in this example.

The total set of components available consists of one disk drive, two memory cards, and four processors, as illustrated in Figure 1. With seven components, there are 128 possible compositions, but they fall into only 30 symmetry classes because components of the same type are interchangeable, so that only the number of components of each type is significant.

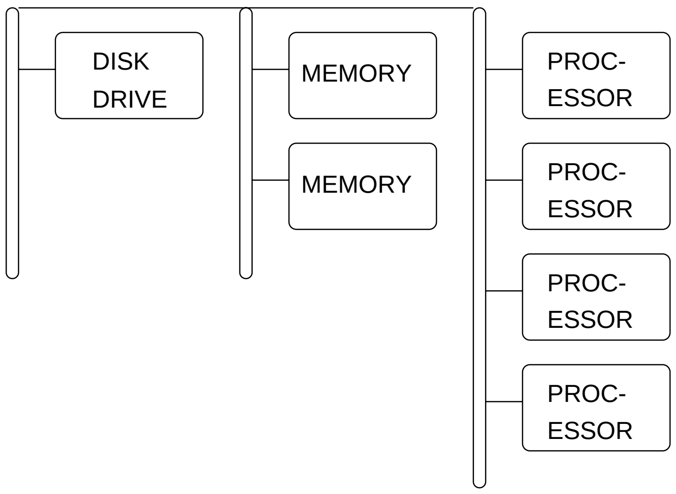

FIgure 1. Example System Components

Suppose that there are two services provided by the system: computation, which requires a processor and a memory card, and logging, which requires a processor and a disk drive. One, both, or neither of these services may be available depending on the current configuration and on which components are operational. To define the set of realizable states, it was assumed that no component sharing was possible.

Our objective is to determine, for each type of component, what are the minimal sets of other components that can replace it in the maximal safe reconfiguration policy. To do this, for each component x, we first construct the flow closure g, and then identify minimal compositions u such that $x \in g(u) - u$.

A Prolog program was written to compute the flow closure, using a routine to compute λ_S as defined earlier. It took advantage of the symmetries inherent in the example by finding the closure for a set of 30 representative compositions. Then, for each type of component – processor, memory card, and disk drive, the minimal replacement set for that component (not containing that component) was found by examining the list of closures.

The results were:

- the disk drive is not replaceable;
- a memory card is replaceable by the other memory card;
- a processor is replaceable by a set of two other processors (not just a single processor!).

Why can't we replace a processor by just one other processor? Because this is a maximal safe policy, having the local reconfiguration property. In such a policy, a replacement rule has to work irrespective of context, i.e., regardless of what state the system is currently in. This is a very constraining condition, and it applies to every replacement rule. If there were a replacement rule saying that a given process is replaced by a given alternate processor, it would fail (to be service preserving) in states in which the chosen alternate processor is already in use. Having a larger replacement set ensures that at least one is available, assuming that all processors in the replacement set are in working order.

Saying that a processor is replaceable by a set of two processors could also be interpreted as offering a choice between them, if only one is necessary. At the moment, the theory does not indicate when some subset of a replacement set will do. We may be able to work out a refinement of that kind in the future.

6. Conclusion

We have begun to investigate what kinds of general statements can be made about the survivability of a fault-tolerant system, given only an abstract description of system services in terms of which sets of components support them and how they can be shared.

So far, this activity has led to the discovery of an intriguing connection between this model and the concept of aggregation in a secure database. Specifically, there is a type of reconfiguration policy analogous to a "safe" information flow policy. In order to apply the security results, we had to find the best interpretation of "sensitivity level" for a set of components.

The definition we found is both necessary and sufficient to ensure that safe flows are both service-preserving and correspond to realizable reconfigurations. Furthermore, the aggregation property of a safe flow policy becomes a localizability condition on reconfiguration rules, permitting failed sets of components to be replaced on the basis of single-component rules.

In order to apply the Meadows result characterizing the maximal safe flow policy, we found ways to obtain some computational savings when generating such policies, although our best algorithm is still exponential-time.

It is not clear whether the local reconfiguration property outweighs the rather restrictive nature of these reconfiguration policies. More general statements about which reconfigurations are permitted within certain classes of systems are certainly possible. We may find that the Meadows conditions for "safety" are stronger than needed. In particular, a reconfiguration rule does not have to be service-preserving in every state. Locality is not really necessary, although additional restrictions beyond the service-preserving condition seem necessary to achieve computational savings.

We hope that this model will lead to other insights about survivability, such as possible ways to measure survivability, and we are looking at a model with a recursive structure, where a component is actually a service at a lower level, with its own components.

References

[1] J. Crow and J. Rushby. Model-based reconfiguration: toward an integration with diagnosis. *AAAI-91*, July 1991, Volume 2, pp. 836-841.

[2] D. Denning. A lattice model of secure information flow. *Communications of the ACM* 19(5), May 1976, 236-243.

[3] G. Friedrich, G. Gottlob, and W. Nejdl. Physical impossibility instead of fault models. *AAAI-90*, July 1990, Volume 1, pp. 331-336.

[4] T. Hinke. Inference aggregation detection in database management systems. *1988 IEEE Symposium on Security and Privacy*, 96-106.

[5] D. Katsuki, E. Elsam, W. Mann, E. Roberts, J. Robinson, S. Skowronski, and E. Wolf. Pluribus–an operational fault-tolerant multiprocessor. *Proc. IEEE* 66(10), October 1978, pp. 1146-1159.

[6] N. Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996.

[7] C. Meadows. Extending the Brewer-Nash Model to a Multilevel Context. *1990 IEEE Computer Society Symposium on Research in Security and Privacy*, pp. 95-102.

[8] R. Negrini, M. Sami, and R. Stefanelli. Fault tolerance techniques for array structures used in supercomputing. *Computer* 19(12), Feb. 1986, pp. 78-87.

[9] P. Neumann. *Computer Related Risks*. Addison-Wesley, 1995.

[10] P. Neumann. Practical Architectures for Survivable Systems and Networks: Phase-One Final Report. SRI International Computer Science Laboratory, Menlo Park, California, January 28, 1999. Online at URL http://www.csl.sri.com/˜neumann/arl-one.ps.