A hierarchic architecture model for dynamic reconfiguration

Conference or Workshop Item

A Hierarchic Architecture Model for Dynamic Reconfiguration

Michel Wermelinger

Departamento de Informática, Universidade Nova de Lisboa
2825 Monte da Caparica, Portugal
E-mail: mw@di.fct.unl.pt

Abstract

Dynamic reconfiguration is the ability to modify a parallel or distributed system while it is running. We adopt the framework developed by Jeff Kramer and colleagues at the system architecture level: changes must occur in a consistent state, which is brought about by "freezing" some system components. The goal is to reduce system disruption, i.e., to minimize

1. *the part of the system to be "frozen" and*
2. *the time taken by reconfiguration operations.*

Towards the first goal we take a connection based approach instead of a component based one. To reduce time, we refine the reconfiguration algorithm by executing changes in parallel as much as possible. Our model also handles hierarchic systems.

1. Introduction

Most systems must undergo several modifications during their lifetime in order to cope with new human needs, new technology or a new environment. Large distributed systems can be described as a configuration of separate, interconnected components. Modifications can therefore occur both at the component level (change implementation, add new functions, etc.) and at the architecture level (add or remove components or connections). We deal only with the latter, in particular we address the following questions:

1. What kind of modifications can be done?
2. How are they performed?

The answer to the first one is given by a configuration model that defines the system architecture and the change process. The second question is about how the changes will be executed by the underlying operating system.

For economical or safety reasons, some systems cannot be stopped or taken off-line to perform those changes. Thus changes are done while the system is running. This is called dynamic reconfiguration and applies only to distributed or parallel systems because centralized "single-thread" systems must be completely stopped to be altered. Normally changes may not be executed at once. For example, to remove a component first it must cease all interactions with its neighbour components. Thus a further question must be addressed:

3. When may the changes be performed?

The answer is: when the components to be changed are in a consistent state. The definition of "consistency" will be given by the model and it is brought about by "freezing" a part of the system which may include components or connections that will not be modified.

To handle the previous questions we adopt a framework developed by Kramer and colleagues [5, 6]. It is simple and general, both in terms of the changes it allows and in terms of the assumptions it makes on systems. Upon closer analysis of the two algorithms proposed for finding the set of system components to "freeze" [6, 2], we have found that neither is minimal regarding the disruption it causes to the system. Switching to a connection based approach we come up with a conceptually very simple yet effective minimal solution.

However, that only accounts for disruption in terms of "size", i.e., what parts of the system are "frozen". It does not take into account for how long they are inactive. Since we work with an abstract, implementation-independent reconfiguration model, our solution just provides an execution order for the change commands such that they are performed as much in parallel as the logical dependencies between them allow.

The third contribution of this paper is the treatment of hierarchic systems, whose components can be made of interconnected subcomponents. For practical purposes, the original work [5, 6, 2] only deals with flat systems. The hierarchic reconfiguration management method to be introduced

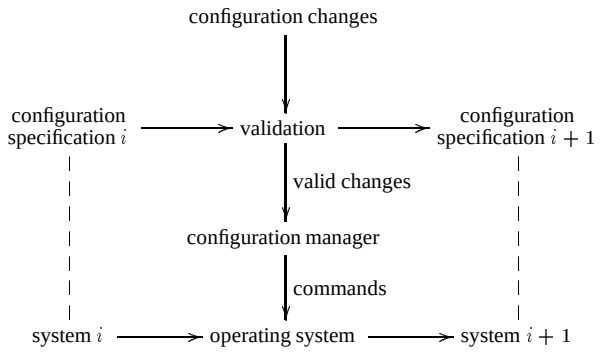

Figure 1. The dynamic reconfiguration model

allows the parallel execution of change commands in different subsystems while taking into account any dependencies among them. Furthermore the method is as modular as the system it is applied to.

The structure of the paper is as follows. The next section summarizes and analyzes the work done by Jeff Kramer and colleagues, showing its problems regarding disruption minimization and hierarchic systems. Whereas the original framework is described mainly in informal terms [5, 6], this paper will provide formal definitions: section 3 describes the refined architecture model and section 4 deals with disruption minimization. Towards that end, it presents a connection based approach and an ordering of change commands. Finally, section 5 describes a method to construct a configuration manager for a given hierarchic system. The last section presents some concluding remarks and future work.

2. The original model

We adopt the configuration model developed in [5, 6] and summarized in Figure 1[1]. In the following we describe the assumptions made by the model for each element appearing in the diagram.

A system can be depicted as a directed graph whose nodes are the system components and whose arcs are transactions between components. As the model is component-based, it assumes there is at most one connection between any pair of components. An arc from a node N to a node N' states that the transaction is initiated by N, although during the transaction communication flow can occur in both directions. Transactions complete in bounded time and the initiator is always informed of completion. In particular, the system does not get into any deadlock or livelock situation. These assumptions will help to prove that the consistent state can be reached in finite time and that the configuration manager will know when. A transaction t is *dependent* on the *consequent* transactions $t_1, t_2, \ldots$ (written $t/t_1t_2\ldots$), if its completion depends on the completion of all the other ones. Otherwise a transaction is called *independent*.

Changes to a system are specified using four commands, to be executed by the operating system, with obvious meanings: `create` N, `remove` N, `link` N `to` N', `unlink` N `from` N'. Given a specification of the current system configuration and the specification of the configuration changes, the validation process checks whether the changes may be (totally or partially) applied to the system and produces the specification of the resulting system. Checks may range from simple syntactic ones (e.g., `remove` N is incorrect if N does not exist in the system) to deep semantic results (e.g., will the resulting system be deadlock free?). In the following it is assumed that changes are valid and that the specification is declarative, i.e., the change commands are not in any particular order.

Given the valid changes, the configuration manager generates the instructions for the operating system to reconfigure the current system, such that the resulting one will conform to the specification produced by the validation process. In particular, the manager performs the following steps:

1. Compute from the change specification the nodes that must be in a consistent state for reconfiguration to take place.
2. Compute the nodes that must become "frozen" in order to achieve consistency over the set of nodes obtained in the previous step.
3. Send a "freeze" message to each node obtained in step 2 and wait for all the acknowledgments.
4. Instruct the operating system to execute changes in the following order: `unlink`, `remove`, `create`, `link`.
5. Instruct the created and the "frozen" nodes (except the removed ones) to resume processing.

There are two approaches based on this model that differ only in steps 2 and 3. The first one [6], which we will call the passive approach, "freezes" a node by preventing it from initiating any new transaction; the second one [2] completely stops the node's execution and therefore will be called the blocking approach.

2.1. The Passive Approach

In this method [6] the "frozen" state is called *passive* and the "freeze" message is `passivate`. To facilitate exposition, let us first handle only independent transactions.

[1] Figures 1 to 5 are adapted from [5, 6, 2].

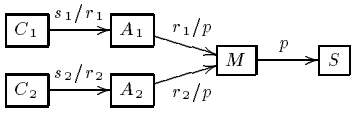

Figure 2. A client-server system with dependent transactions

A component is passive if it is not engaged in transactions it initiated and if it will not start new ones. However, it must accept and service transactions in order to let other nodes become passive. Therefore, passiveness is reachable in finite time: a component just has to wait for the transactions it initiated to finish (this is guaranteed to happen) and then make sure it will not start new ones. The passive state is just a necessary condition for reconfiguration. In order to guarantee a consistent and stable internal state, in addition to being passive a node should not have any outstanding transaction to service. This is called *quiescence* and depends on those components that can initiate transactions with the node. Therefore, the passive set of a node Q, $PS(Q)$, is defined as Q and all nodes with connection arcs towards Q. It is easy to see that Q is quiescent if all nodes in $PS(Q)$ are passive.

The quiescent set QS for a given change specification is the set of nodes that must be quiescent during the reconfiguration, namely those that will be removed and the initiators of transactions that will be added or removed. Newly created nodes are automatically quiescent. The set of nodes to "freeze", called change passive set, is then simply $CPS = \bigcup_{i \in QS} PS(i)$.

To see why this does not work for dependent transactions, consider a system with clients C_i accessing through agents A_i a server S managed by M (Figure 2). If the server is going to be replaced, then both S and p will be removed. Thus the configuration manager calculates $QS = \{M, S\}$ and $CPS = \{A_1, A_2, M, S\}$. However, if a client has a new request s_i, then the respective agent cannot service it because according to the definition of passiveness it may not initiate r_i (on which s_i depends). This would lead to a partially incomplete transaction, i.e., to an inconsistent state of the whole system during reconfiguration. On the other hand, allowing A_i to start transaction r_i would lead to new transactions on the manager and on the server, which therefore would not be in the quiescent state.

To solve this problem, the notion of passive state must be changed. Otherwise reachability of the quiescent state in bounded time would be lost. If A_1 is to be replaced, then $QS = CPS = \{C_1, A_1\}$. If A_1 becomes passive before C_1, and C_1 just initiates a new transaction s_1 before getting the `passivate` command from the configuration manager, then the client will never become passive because r_1 is not initiated. In this case one could order the commands (`passivate` C_1 before `passivate` A_1), but for systems with mutual dependencies like the one in Figure 3 no such ordering is possible.

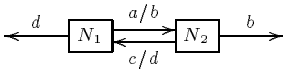

Figure 3. Mutual dependencies

The notion of passive set must also change, since the nodes that may initiate transactions with a given node are not just its immediate neighbours. The new definitions are thus as follows.

- In the *generalized passive state* a node is not engaged in non-consequent transactions it initiated and it will not initiate new ones. Furthermore the node accepts and services all requests, initiating consequent transactions if necessary.
- The *enlarged passive set* of a node Q, $EPS(Q)$, includes Q and all nodes that can initiate transactions which result in consequent transactions on Q.

Notice that both definitions reduce to the old ones in case all transactions are independent. The reconfiguration algorithm remains the same, except that $PS(i)$ is substituted by $EPS(i)$ in the calculation of CPS.

The server replacement in Figure 2 is now correctly handled. Since $EPS(S) = \{C_1, A_1, C_2, A_2, M, S\}$, all nodes have to be passivated. Even if all components but C_1 are already passive, any pending s_1 transaction will be serviced (through A_1 and M) by the server and therefore the client can become passive and reconfiguration may start.

In general, systems are not flat as assumed until now but hierarchic, i.e., some nodes (called composite) are made of connected subnodes. A composite node is connected to other nodes through some of its subcomponents. The transaction dependency of a composite component must be derived from its subcomponents. The following substitution rule is given in [6]:

> "when composing 2 nodes, substitute the consequents for each occurrence of the dependent transaction which is hidden by the composition."

The rule can be iterated on components and connections (Figure 4). To simplify reconfiguration management, [6] suggests that a composite node is considered to be passive if all its subnodes are, and that all transactions between composite nodes are independent.

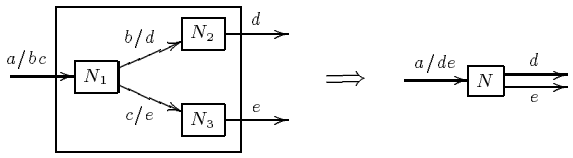

Figure 4. Composing dependencies

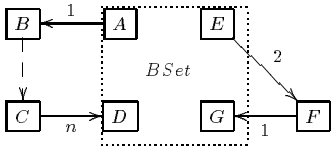

Figure 5. A blocking problem

2.2. The Blocking Approach

An alternative method is presented in [2]. It assumes that a node is consistent and self-contained except during transactions, as those are the only interactions with the outside environment. Thus, to make a node quiescent it is enough to block it while it is idle (not engaged in any transaction). A component is also assumed not to interleave transactions: while handling a request a node may not service any new one, even if it comes from a different connection, and it may initiate only consequent transactions. This is used to prove that the blocked state is reachable in finite time.

The basic algorithm is thus to send a `block` message to the nodes in the quiescent set (called BSet, short for blocking set, in [2]). As soon as such a node N is idle, it blocks and sends an acknowledge to the configuration manager. Since some of the nodes that depend on N may also have to block, N must temporarily unblock to service some requests. However, it must be guaranteed that at some point no more such requests will arrive and N will remain blocked. The basic question is therefore: what transactions should a blocked node service?

It is obvious that it cannot process just any incoming transaction, since it might come from a node that is not affected in any way by the reconfiguration and as such might initiate a new transaction any time. Thus the blocked node would have to unblock unpredictably and the safe state needed for reconfiguration to begin would never be reached. It is also evident that at least the transactions initiated by other BSet members will have to be serviced in order for them to become blocked. On the other hand, not every request from a non-BSet member can be ignored. Consider the cases depicted in Figure 5. Node D must service the request from C because it is the nth consequent transaction of a transaction initiated by A, which must be completed for A to become blocked. In the second case on the right half of the figure, component F has initiated a transaction with G before getting a request from E. IF G does not service the transaction, F will never be able to start attending E's request since transactions do not interleave.

One could let BSet nodes unblock just in those situations but the authors think this is non-trivial and has great runtime overhead. Instead they propose the BSet to grow dynamically in step with outgoing transactions. When a node gets a request from a BSet member, it becomes a member too, and only requests from BSet members are attended; all other are queued and serviced after the reconfiguration. In the previous cases, it means that the BSet grows to encompass the whole system, and therefore D and G will service the transactions initiated by C and F respectively.

Notice that the BSet has two kinds of members: those that "really" must block due to the reconfiguration and those that block in order to let members of the first kind to get blocked. Therefore a distinction is made between the original BSet and the extended BSet. Their union is the BSet. When all the original BSet nodes become blocked, the components in the extended BSet can be unblocked since their *raison d'être* has ceased. The disruption thus first grows and then shrinks.

As the calculation of the BSet is dynamic, the reconfiguration algorithm is distributed through the configuration manager and the nodes. Each node runs the same code in a transparent way using hooks that are called upon relevant events like message arrival (from another node or from the configuration manager), transaction begin and transaction end. The application programmer only has to mark in the component's code where the last two hooks have to be activated. The code run by each hook basically updates local variables and sends messages to other nodes or to the configuration manager when necessary. The main variables and messages are those that concern the BSet. The configuration manager computes the original BSet (in the same way as the passive approach, since it is the quiescent set) and the so called PSet (short for primed set), the set of all nodes that may be recipients of transactions initiated by BSet members. Whenever such a node receives a message from a BSet member, it informs the configuration manager that it must be added to the (extended) BSet. The configuration manager calculates the new PSet and sends the updated value of the BSet to all those nodes.

3. Discussion

The authors of the approaches just described discuss their results, but since [2] contains not a single example, comparison with [6] is stated in brief and vague terms. Since we

use the same framework, we take a closer look at the two different methods in order to gain a better insight into the reconfiguration process to achieve further reduction in disruption. Both approaches will be analyzed in terms of disruption, run-time overhead, implementation, and how they deal with hierarchic systems.

We adopted this model because, as stated in [2], other work in dynamic reconfiguration either only deals with tool and language support to describe and execute the changes [1] assuming that the system is already in a safe state, or it imposes limitations to guarantee maintenance of the system's state [4, 3, 9]. Limitations can be on the kind of system (e.g., only client-server) or on the kind of changes handled (e.g., a component may only be replaced by a specialization of it). Sometimes they are due to the existence of special mechanisms to capture and recover the application state prior to the change. The described framework is more general: it allows any kind of changes, it provides a means to achieve the stable state, and it does not require special mechanisms.

3.1. Implementation

In the first approach the application programmer is expected to provide code that will allow the component to reach the passive state and keep it, whereas in the second approach this happens transparently due to the added assumption that nodes are consistent when there are no interactions with the rest of the system. Of course, the other side of the coin is that the implementation is hidden away into the hooks which must trap low level events like message arrival. This means that the implementor of this approach must have access to the operating system source or else program a new layer that will be used by the applications.

On the other hand, the first approach requires just one `passivate` message for each node in the extended passive set, while the second method generates initially $|PSet_0|$ messages and then $1 + |PSet_i|$ messages for the addition of the ith member of the extended BSet. Also, the messages in the static method are just tokens, while those in the dynamic approach are descriptions of (potentially large) sets. The blocking approach has thus much greater run-time overhead and is more complex to implement, but imposes less burden on the component programmer.

3.2. Disruption

The important point in any dynamic reconfiguration method is that the "freezing" of a node N does not prevent other nodes from reaching their "frozen" state. Basically, both approaches solve the problem by "freezing" also every node that depends on N or on which N depends. This does not minimize disruption and in fact may involve many components besides those that will be affected by reconfiguration.

To illustrate the differences between both approaches, let us apply them to common examples. We will write OBS and EBS for the original and the extended BSet, respectively. The first example is the system of Figure 2. If the server is to be replaced, we have seen that the first approach passivates all nodes. The second method considers $OBS = \{M, S\}$, $EBS = \{\}$ and $PSet = \{S\}$, because the only outgoing transaction from a BSet member is p and goes to the server. This means that on occurrence of p, $EBS = \{S\}$ and on its completion both M and S block because they are idle and in the BSet. Notice that the clients are not blocked and therefore may initiate new transactions during reconfiguration. Since M is blocked, it will queue the requests and service them after the changes done to the system. This was considered an inconsistency in the passive approach, but in our opinion this is perfectly acceptable because the server manager has not been changed. Therefore its interface with the agents and the new server is the same as previously. This means that the new server is able to attend requests sent to the old one. To sum up, the blocking approach causes less disruption. The reason is that a passive node (and in particular a quiescent one) is active regarding transactions it services. Therefore, to achieve inactivation the node must be "shielded" from outside requests and that shield (the extended passive set) must remain during reconfiguration. No such shield is required in the second approach since the components actually stop.

Now consider the same system but where the first client has to be replaced. In this case $QS = EPS = \{C_1\}$ since no component depends on C_1. Therefore, as soon as the transaction terminates, C_1 will become passive and automatically quiescent. Reconfiguration can start, while all other components remain active. Applying the blocking approach one has $OBS = \{C_1\}$, $PSet_0 = \{A_1\}$, $EBS_0 = \{\}$. If there is a pending client request, $EBS_1 = \{A_1\}$ and $PSet_1 = \{M\}$. Since the transaction is dependent, after two more steps $EBS_3 = \{A_1, M, S\}$. In other words, to replace a client, the server is blocked (even if temporarily)! This would remain so even if all transactions being executed were independent. In this example the first approach causes much less disruption, contrary to the claim in [2] that the blocking approach performs always at least as well as the passive method. The reason is that the blocking approach is purely dynamic: it does not precompute the dependencies between nodes, which is essential to determine whether the blocking of two components will interfere with each other. Therefore at run-time the method goes through *all* the nodes an OBS member depends on, which form the EBS. If the configuration manager would compute the paths between OBS members, disruption could be greatly reduced in most cases.

The last example is the left half of the system in Figure 5. In the second approach, if A is not engaged in any transaction with B, it will block immediately. Thus as soon as D is idle it will get blocked too and reconfiguration starts. In the first approach, all nodes from B to C will be passivated even if no dependent transaction will occur. This is the advantage of a dynamic method. It only takes into account transactions that are actually occurring in the running system, while a static analysis must involve all transactions that *may* occur.

The authors have concentrated on the number of nodes that are passivated or blocked by their methods, but we think that indirect disruption must also be taken into account. Since a blocked node does not any processing whatever, any transaction it services or initiates unrelated to the reconfiguration will also be stopped and that may lead to (partial) inactivation of other components. Since passive nodes still service requests they cause indirect disruption in smaller scale. But internal processing that requires initiation of transactions is still hindered. This is recognized in [6]. The authors observe that the replacement of the server in Figure 2 passivates the clients thus stopping them from interacting with other nodes not shown on the figure. Therefore, they should only be passive with respect to the server being replaced, not other nodes unrelated to the change. This could be achieved by distinguishing the relevant connections and modeling their state (connected-passive, connected-active, disconnected). This would allow more granularity, but the authors think it would lead to more complex substates and more complex actions to obtain consistency since the nodes would be partially active. Therefore they think their approach, while not minimal, is simple and sufficient.

In our opinion there is another factor that contributes to a greater disruption than necessary in some cases: the requirement for quiescence of the initiator node in (un)link changes.

Let us assume that the change specification contains a command `unlink` N `from` N' for a nonconsequent transaction. It is not necessary for N to be quiescent. It is enough to be passive, thus not starting any new transaction with N'. Consider the right subsystem of Figure 5 where connection 1 will be removed. If $F \in QS$ then E and every node that depends on transaction 2 would be in EPS. Therefore they and all nodes on which they can initiate transactions would be partially inactive. If F were only passivated, the extended passive set would not include the other nodes, reducing direct and indirect disruption greatly.

Also, if there is a `link` N `to` N' command but neither N nor N' are changed, then the new connection is the replacement of a previously existing connection or it is an optional connection (because N was already working without it). In any case N does not have to be quiescent or even passive. It is only in those states if it has to be replaced or if some of its connections will be removed. In our opinion, the addition of connections by itself should not impose passiveness.

Both approaches measure disruption only in terms of nodes, neglecting the time factor. In the configuration model described in section 2 first components are "frozen", then change commands are applied, and finally components are activated. This does not minimize disruption time because each phase can only begin after the previous one ended. Moreover, commands are performed in a fixed sequence (first all `unlink`, then all `remove`, etc.). It is obvious that in many cases some changes are independent of others. In those cases a part of the system might be changed without having to wait for nodes in other parts to be "frozen", or commands of different kinds might be performed in parallel.

3.3. Hierarchic systems

In [2] no reference is made to hierarchic systems. In fact, the blocking approach does not work for them since a composite node will in the general case interleave transactions, because its subcomponents run in parallel. As written before, [6] deals with such systems but their treatment is still very sketchy. Basically, it only indicates how to compute a composite node's dependencies from its subcomponents. From there the extended passive set at the higher level can be computed. If the composite node has to be passivated, all its subcomponents should. This certainly does not minimize disruption. We also feel that requiring independent transactions between composite components (as in CONIC [8]) to reduce the number of those to be passivated may lead to extremely large components or to many small ones. In any case it may force the system designer to partition the system into artificial composite components that are uneasy to work with. But more importantly, [6] does not deal with the interaction between changes at different levels or how changes at a lower level will affect higher levels of the component hierarchy.

4. The refined model

Although the original analysis of the requirements for dynamic configuration [5] stressed the importance of modularity and well-defined component interfaces, the model presented in [6] does not support it. However, the concrete configuration language presented in [5], CONIC, and its successor DARWIN [7] provide a mechanism to specify the communication points of a component, called ports. Our model will thus support that notion, too. An interface is just a set of ports, each being used either to initiate transactions or to receive requests. Since the environment has no access to the inner structure of a component, the programmer must provide in the interface the dependencies between initiator ports and recipient ports.

Definition 1 A *node interface* is a triple $\langle I, R, D\rangle$ where

- I is the finite set of *initiator ports*;
- R is the finite set of *recipient ports* such that $I \cap R = \emptyset$;
- $D \subseteq R \times I$ is the *port dependency* relation.

A system is simply a set of connected nodes, where a connection is given by an initiator port and a recipient port. To capture sound software engineering principles (modularity, encapsulation, data hiding, etc.), a system has no access to the inner structure of its nodes; it knows only their interfaces.

The original model is intended for node based reconfiguration, i.e., "freezing" is done upon nodes. Moreover, the computation of the passive and blocked sets depends only on the pattern of connections, not on their number. Therefore the model can assume without loss of generality that there is only one arc between a given pair of components. A connection based approach like ours distinguishes individual transactions and thus one must allow several connections to be linked to the same port (but only one transaction for any given pair of ports). This covers typical situations like client-server (all client transactions linked to same server recipient port) and broadcast (many transactions with common initiator port). To avoid deadlock, the connections (together with the port dependencies) may form no cycle. Formally, there may be no closed sequence of alternating recipient and initiator ports such that every initiator port depends on the succeeding recipient port which in turn is linked to the next initiator port.

Definition 2 A *system* is a pair $\langle N, T\rangle$ where

- N is a non-empty finite set of node interfaces;
- $T \subseteq \bigcup_{n\in N} I_n \times \bigcup_{n\in N} R_n$ is the set of *transactions*.

A *non-empty path* is a sequence of ports $r_1 i_1 r_2 i_2 \cdots r_m i_m$ with $m > 0$ such that

- $\forall j \in \{1, \ldots, m-1\}\ \langle i_j, r_{j+1}\rangle \in T$;
- $\forall j \in \{1, \ldots, m\}\ \exists n \in N\ \langle r_j, i_j\rangle \in D_n$.

For every non-empty path $r_1 \cdots i_m$ one has $\langle i_m, r_1\rangle \notin T$.

The original model assumes that the dependencies among transactions are given with the system. We feel that our notion of port dependency is more realistic and more flexible since it allows the system architect to work with components he has not programmed himself. Besides, it is a more primitive notion because the dependencies among connections can be computed from those between ports: if recipient port r depends on initiator port i, then *any* transaction received by r starts a transaction (i.e., depends) on *every* connection from i. As in the original model, the inverse is not true: the component might start a transaction on port i without having received any request on port r. The transaction dependency relation is closed under transitivity.

Definition 3 Given a system $\langle N, T\rangle$, the *transaction dependency* relation $/ \subseteq T \times T$ is defined as $\langle i, r\rangle / \langle i', r'\rangle \Leftrightarrow (\exists n \in N\ \langle r, i'\rangle \in D_n) \vee (\exists t'' \in T\ \langle i, r\rangle / t'' \wedge t'' / \langle i', r'\rangle)$.

A transaction t is *dependent* (on the *consequent* transaction t') if $\exists t' \in T\ t/t'$, otherwise t is *independent*.

The acyclic condition on port paths can thus be restated as: transaction dependency is anti-reflexive.

Proposition 4 *In a system* $\langle N, T\rangle$, $\nexists t \in T\ t/t$.

To build modular architectures it must be possible to abstract systems into nodes which will be part of other systems. A system will be encapsulated in a composite node by hiding part of the system's ports. The dependencies of the remaining visible ones (i.e., the ports of the composite node) are given by the underlying system.

Definition 5 A *composite node* consists of an interface $\langle I, R, D\rangle$ and a system $\langle N, T\rangle$ such that

- $I \subseteq \bigcup_{n\in N} I_n$;
- $R \subseteq \bigcup_{n\in N} R_n$;
- $D = \{\langle r, i\rangle \in R \times I \mid r i_1 \cdots r_m i$ is a non-empty path in $\langle N, T\rangle\}$.

If a node is not decomposed into further nodes, then it is called *simple*. Formally, only its interface is available. A system is *hierarchic* if it contains at least one composite node. Strictly speaking, given a system it is impossible to know for any of its nodes whether it is simple or composite because the formal definition of a system only provides the node interfaces. Thus it is possible for a simple node to be changed into a composite one and vice-versa in a transparent manner to the system.

5. Minimizing disruption

From the long summary and analysis of the passive and blocking approaches it becomes clear that to minimize disruption we must look for a static blocking method at the connection level. To ensure that blocking a connection will not prevent others from reaching the blocked state, we can use a previously mentioned idea: to order the execution of

"freeze" commands. This works at the connection level because transactions do not form cycles. Extending the execution ordering to all commands one can define precisely what changes may be performed in parallel to reduce disruption time.

5.1. The connection approach

The essence of our proposal is to block only those connections that will be removed. To block a connection its initiator node waits for any ongoing transaction (on that connection) to finish and then simply does not start a new one. For this to work we assume, as in the original model, that a transaction finishes in finite time and that its initiator knows when it ends. A simple implementation might be the following. For each component, assign to each transaction T_i it might initiate a boolean variable `blocked[i]` initialized to false and a semaphore `S[i]`. Then substitute the transaction code T_i by

```
P(S[i]);
wait while blocked[i]; Ti;
V(S[i]);
```

and add the following case to the code that dispatches the incoming requests:

```
if msg.command = block then begin
  i := msg.arg; P(S[i]);
  blocked[i] := true; V(S[i]);
  send(config_manager, blocked, i)
end
```

This code can also be provided by three hooks if wished. One to be called on transaction begin, one on transaction end. These hooks must be explicitly called by the component's programmer, passing the transaction identifier as argument. The third hook would be called transparently to the component on message arrival. Compared to the blocking approach, run-time overhead is small since only one simple `block` message per connection is sent and acknowledged. However, the number of messages is usually larger than in the passive approach because each node to be removed has to receive as many `block` messages as the connections it has.

Blocking a connection means that the node will not service any transaction that depends on the blocked one. To ensure that the blocking of one connection will not prevent other pending transactions to block, the configuration manager orders the blocking according to dependency: if transaction t depends on t' then the `block` message is sent to the initiator of t' only after t is known to be blocking. This is always possible because transactions do not depend cyclically on each other.

Consider again the client-server system of Figure 2. Let us assume that C_1 and the server will be replaced. Then s_1 and p must block because they will be removed, but p cannot simply block at once because it may have to service a pending s_1 request (or else s_1 could never terminate and get blocked). Therefore, blocking s_1 before p we are sure that the blocking state is reachable for each link. Also, any request received by manager M after p blocked can be safely queued until the server has been replaced because it is known that any connection that depends on p and that had to block has already done so.

Notice that this method would not work if the server would be allowed to be simply removed without being replaced by a new one. In that case a partially completed s_2 request could remain after reconfiguration: clearly an inconsistent state. We assume that the validation process has ruled out such cases. If a consequent transaction is removed, either a replacement connection is created or else the transactions which depend on the removed one are changed too.

The original reconfiguration model distinguished two kinds of commands: those that are given in the change specification (`create`, etc.) and those that are used to "freeze" the components (`passivate`, `block`) and to activate them again after reconfiguration end. The former are common to the passive and blocking approaches, while the latter are specific to each approach. In our model the "freeze" command blocks a connection and we will ignore the activation commands since they are not fundamental for the main issues of this paper, namely disruption minimization and hierarchic systems. Furthermore, as multiple transactions are allowed between the same pair of components, the syntax of the `(un)link` commands has to change slightly.

Definition 6 A *command* is either of `create` n, `remove` n, `link` t, `unlink` t or `block` t, where n is a node interface and t a transaction.

5.2. The partial order

To minimize disruption time, the precise execution of the commands issued by the configuration manager is given by a temporal order $<$: if $c < c'$ then command c' can only be executed after command c has completed. Commands that are not related through the ordering can be executed in parallel. It is obvious that the order must include the following relationships:

1. If a transaction t depends on a transaction t', then t must be blocked before t'.
2. A connection must be blocked before it is removed.
3. A node can only be removed after its connections have been removed.

4. A node can only be linked after its creation.

We will be conservative and impose a further restriction. In some systems it might not be necessary and thus further parallelization can be achieved. Consider a simple system with a client linked through transaction c to a server. If the server is to be replaced then a new connection c' is needed. However, since the client remains the same, the communication protocol with the new server is the same as with the old one. Therefore, c' is the same transaction as c and we feel it does not make sense to link c' before unlinking c. Besides, it might lead to execution errors if the implementation of the client assumes that there is always only one connection on that particular port. The general rule is:

5. A transaction initiated by a node can be established only if no more transactions initiated by it will be removed.

As can be seen by exhaustive inspection of all possible interactions between the existing kinds of commands (`block`, `link`, `unlink`, `remove`, `create`) no further rules are necessary since there are no other dependencies between the commands and thus they may run in parallel.

Definition 7 Given a set of commands C for a system $\langle N, T\rangle$, the *command order* $< \subseteq C \times C$ is the smallest relation that satisfies

1. `block` $t <$ `block` t' if t/t' and $\nexists$`block` $t'' \in C$ $t/t'' \wedge t''/t'$;
2. `block` $t <$ `unlink` t;
3. `unlink` $\langle i, r\rangle <$ `remove` n if $i \in I_n$ or $r \in R_n$;
4. `create` $n <$ `link` $\langle i, r\rangle$ if $i \in I_n$ or $r \in R_n$;
5. `unlink` $\langle i, r\rangle <$ `link` $\langle i, r'\rangle$.

Since the configuration manager will directly implement the command order, it is desirable to avoid redundancy. Therefore the ordering is an immediate precedence relation: if $c < c'$ then there is no command c'' such that $c < c'' < c'$. Due to the nature of the five cases this could only happen with `block` commands (case 1). Therefore the definition above imposes the additional condition.

Let us return to the example of Figure 2. Only 4 steps are necessary to replace the first client and the server whereby each step consists of several commands executing in parallel:

1. `create` C_1', `block` s_1, `create` S'
2. `unlink` s_1, `block` p
3. `link` s_1' (the connection from C_1' to A_1), `remove` C_1, `unlink` p
4. `link` p' (the connection from M to S'), `remove` S

Notice that in some cases some commands of step $i+1$ can start without step i being completed (the exact order is given by Figure 6 as explained in the next section). Since our rules take into account the specific components or connections on which the commands are to be executed, the disruption suffered by each system part being changed is greatly reduced since change actions are much more interleaved than in the original model.

To summarize, a connection based approach is not only advantageous in terms of the number of parts being "frozen", but also in terms of minimizing disruption time. In fact, in the node based approaches several nodes have to "freeze" just to let those nodes that really matter for the reconfiguration to become quiescent. In practice this means that reconfiguration can only start after all nodes have "frozen". We think it is possible to have rules that allow one to calculate the exact set of nodes that have to "freeze" for a given change command to be executed, but those rules would be much more complicated than those shown above. Given that in a connection based approach the number of parts to be "frozen" is much smaller, and that "freeze" and change commands can be better interleaved, we conclude that our method can reduce disruption time considerably.

6. The configuration manager

Since a configuration manager executes several commands with some dependencies among them, we observe that such a manager can be seen as a parallel system too, with components and transactions. The goal is to have a precise definition of a configuration manager for a given set of commands to be applied to a given system. In this way the same framework can be used both for managers and the systems they reconfigure. In particular, the definition to be obtained can serve as a basis for a straightforward implementation of configuration managers, although our main goal is to provide a system view of a manager. To facilitate exposition we will start with flat systems.

The basic idea is that each change command is implemented by a component, and connections between components make dependencies between the corresponding commands explicit. To be more precise, if $c < c'$ then the component corresponding to c' will initiate a transaction with the component corresponding to c. The transaction can be seen as a request from c' to execute c. Once the acknowledgment is received, c' can execute. If c' depends on several commands it must wait for all its requests to be attended. A command is executed only once, even if several other commands are connected to (i.e., depend on) it.

A component implementing a command c must therefore have two ports. The recipient port s_c receives all requests

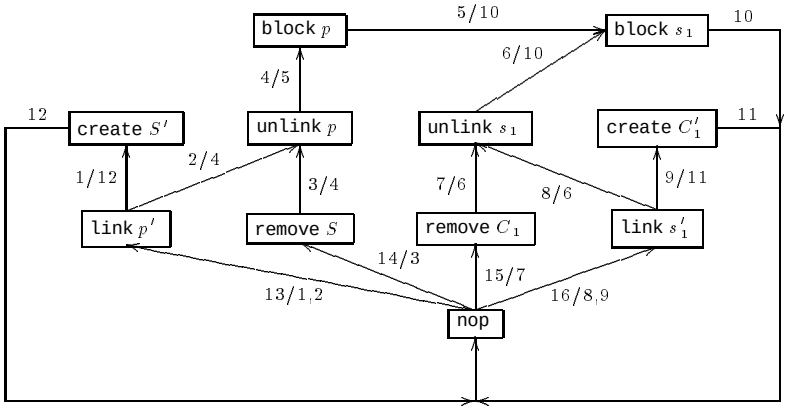

Figure 6. A specific configuration manager

from the successors of c, i.e., those nodes that can only execute after c. The initiator port p_c sends requests to all predecessors of c to start execution and waits for the acknowledgments. It is obvious that s_c depends on p_c.

In some cases a command c does not depend on the execution of others. In other words, there is no c' such that $c' < c$. The inverse can also happen: no c' depends on c. For example, if connections are to be removed, there is always at least one `block` command to be executed first (i.e., it depends on no other one) and at least one `block` to be executed last. In these cases the corresponding components only need one port. Instead of providing special component definitions we take a generic approach. The configuration manager has always one special nop component with one recipient port $s\mathsf{nop}$ and one initiator port $p\mathsf{nop}$ (like any regular component) but there is no dependency between them. For any request received by $s\mathsf{nop}$ an acknowledgment is immediately sent. Likewise, any transaction linked to $p\mathsf{nop}$ is immediately started. To see why this works, consider the case where there is no c' such that $c' < c$. Since c depends on no other command, it can execute at once. In other words, the fact that c has no predecessor can be seen as its predecessor being the "empty" command nop. Therefore, if c's predecessor port p_c is linked to the successor port $s\mathsf{nop}$, the request from c is immediately attended by nop and therefore c can execute at once as wished.

Definition 8 The *system configuration manager* that reconfigures the flat system $\langle N, T\rangle$ according to commands C is a system $\langle N', T'\rangle$ where

$$\begin{aligned} N' &= \{\langle\{p_c\},\{s_c\},\{\langle s_c, p_c\rangle\}\rangle \mid c \in C\} \\ &\cup \{\langle\{p\mathsf{nop}\},\{s\mathsf{nop}\},\emptyset\rangle\} \end{aligned}$$

and T' is the smallest relation that satisfies

1. $\langle p_c, s_{c'}\rangle \in T'$ if $c' < c$;
2. $\langle p_c, s\mathsf{nop}\rangle \in T'$ if $\nexists c'\ c' < c$;
3. $\langle p\mathsf{nop}, s_c\rangle \in T'$ if $\nexists c'\ c < c'$.

According to this definition the reconfiguration of the client-server system of Figure 2 can be done by the manager depicted in Figure 6 which allows us to quickly see the ordering of the commands, in particular which must be executed sequentially and which can run in parallel. Notice that the execution path starts and ends at the nop node, and that the four steps presented in the previous section correspond to the four levels of the topological sort of the graph.

Hierarchic systems pose a problem that does not occur in flat systems: if commands c and c' apply to different subsystems, it might still be the case that $c < c'$ (or vice-versa) due to the way the subsystems are connected. As an example let us consider Figure 7 where the dotted lines indicate for each port of a composite node which is the corresponding port of the contained system. Assume further that for each of B, C, D, and E, the recipient port depends on the initiator port. The same applies to N_2, according to the definitions of composite nodes and transaction dependency, and thus b/d as seen on the right. Moreover, for the given configuration one has a/e, which is not apparent just by looking at N_1. If A and F are to be replaced, those two connections cannot be blocked in parallel. One could flatten the whole system to discover that `block` $a <$ `block` e, but that defeats the whole purpose of building a modular system.

Therefore, to reflect the hierarchy of a system and the benefits of its partitioning, we propose a configuration manager for each node. We have seen that a manager for a system is a system itself. Likewise, the manager of a node will be a node, with an interface that will allow it to be linked to other configuration managers. The problem is thus what interface a node manager needs and how should it be linked to other managers. The goal is to achieve the correct order

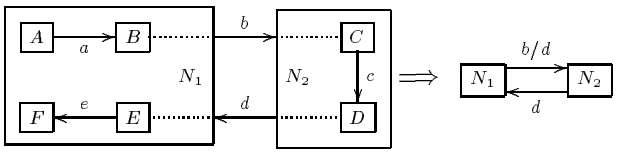

Figure 7. A hierarchic system

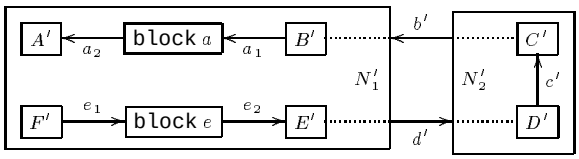

Figure 8. Composing reconfigurations

of command execution in the most modular possible way. In other words, a node manager should only know about the subsystem it manages, not about the managers it is linked to. Therefore, the internal structure of a node manager should be such that it can work in any possible context.

The solution to the problem is based on the following observations. Let us assume that node x has an initiator port i linked to recipient port r of node y. Thus any change inside x that depends on transaction $\langle i, r\rangle$ must occur before the changes inside y that depend on $\langle i, r\rangle$. Therefore the requests of the change commands inside y must be acknowledged by the change commands inside x. Therefore the direction of requests is opposite to the direction of the transaction that establishes the dependency between x and y. To sum up, the configuration manager x' for x has *recipient* port i, the manager y' for y has *initiator* port r, and the requests of y' are passed to x' through transaction $\langle r, i\rangle$. If the system's transaction $\langle i, r\rangle$ is going to be blocked then the reconfiguration manager for the whole system cannot just link sub-manager x' to sub-manager y'. In this case port r of y' is connected to the recipient port of `block` $\langle i, r\rangle$ whose initiator port is linked to port i of x'.

To put it in more general terms, the configuration manager for a hierarchical system S consists of one component for each command, one component called `nop`, and one configuration manager for each node in S. A node manager has the same interface as the node whose reconfiguration it manages, except that initiator ports are exchanged with recipient ports. This implies that the manager's port dependency relation is the inverse of the node's dependencies, and that connections among node managers are the opposite of those between nodes, except for transactions that must be removed. Those will be of course substituted by the respective `block` command.

Definition 9 The *system configuration manager* that reconfigures the hierarchic system $\langle N, T\rangle$ according to commands C is a system $\langle N', T'\rangle$ where

$$\begin{aligned} N' &= \{\langle R_n, I_n, D_n^{-1}\rangle \mid n \in N\} \\ &\cup \{\langle \{p_c\}, \{s_c\}, \{\langle s_c, p_c\rangle\}\rangle \mid c \in C\} \\ &\cup \{\langle \{p\mathsf{nop}\}, \{s\mathsf{nop}\}, \emptyset\rangle\} \end{aligned}$$

and T' is the smallest relation that satisfies

1. $\langle r, i\rangle \in T'$ if $\langle i, r\rangle \in T \wedge$ `block` $\langle i, r\rangle \notin C$;
2. $\langle r, s_c\rangle, \langle p_c, i\rangle \in T'$ if $c =$ `block` $\langle i, r\rangle$;
3. $\langle p_c, s_{c'}\rangle \in T'$ if $c' < c$;
4. $\langle p_c, s\mathsf{nop}\rangle \in T'$ if $\nexists c'\ c' < c$;
5. $\langle p\mathsf{nop}, s_c\rangle \in T'$ if $\nexists c'\ c < c'$.

Definition 10 The *node configuration manager* for a composite node with interface $\langle I, R, D\rangle$ and system $\langle N, T\rangle$, to which commands C will apply, is a composite node with interface $\langle R, I, D^{-1}\rangle$ and system configuration manager $\langle N', T'\rangle$.

Applying the definition to the example of Figure 7 we get the manager depicted in Figure 8. Notice how indeed e gets blocked only after a, since the request of `block` e is passed along N_2' back through the other port of N_1' to `block` a.

For the definition to be complete it remains to be said how a configuration manager x' for a simple node x behaves. As for any node manager its interface is the "mirror" of the node's interface, and the same happens to port dependencies. If recipient port r of x' (the manager!) depends on initiator port i, then x' must forward any request received on r to port i[2]. If r does not depend on any initiator port, then x' acknowledges immediately any request received by r. As usual, all transactions connected to a initiator port are also immediately started.

Returning to our example, let us assume that all nodes from A to F are simple. Then a can be blocked at once since A' attends the request made by `block` a. F' issues a request to `block` e which gets forwarded by C' and D' until A' which gets immediately acknowledged at that point. In a slightly optimized implementation of this model, if `block` a had already executed, it would acknowledge `block` e's request without forwarding it to A'. As a further example, consider that there is no dependency between the ports of C (i.e., b is independent of d)., Then `block` e is acknowledged at the recipient port of C' and therefore a and e can be blocked in parallel as desired.

[2] A composite node manager also does this, the only difference being that the forwarding is done through the dependency path made explicit by the composite node's architecture.

7. Conclusions

Dynamic reconfiguration is a problem specific to parallel and distributed systems that has practical relevance. We have adopted a simple and general framework at the software architecture level stating which parts of the system should be "frozen" in order to achieve a stable consistent state and how the "freezing" and the changes are performed. We analyzed, formalized, refined, and extended the framework in order to minimize disruption and to handle hierarchic systems.

In fact, switching from a component based to a connection based approach, we have come up with a minimal solution (since it only blocks the connections that will be removed) that is conceptually very simple and not harder to implement. On the other hand, for the first time for this framework, we have concentrated on the time taken by the reconfiguration process. In particular we have defined an order for the change commands that may reduce, considerably, the disruption of independent parts of the system being reconfigured. The assumption is, again, that commands may be executed in parallel.

Since a configuration manager executes the commands of a given change specification, it can be seen itself as a system of interconnected components, where a component is a single change command and a connection denotes the dependency between the two commands it links together. This model gives a precise and complete account on how a configuration manager may execute a change specification. The model is also particularly useful for hierarchic systems, showing how the reconfiguration process of the whole system can be obtained simply by connecting the configuration managers of the subsystems together, in a way that mirrors the connections between the subsystems.

We plan to further develop this view of a configuration manager as a system like the one it manages. In particular, this view implies that a configuration manager might be subject to reconfiguration too. In other words, a change specification can be changed. We think this might be useful in two situations: failures and validation. If an ongoing reconfiguration fails for some reason, then one may try to find another but equivalent (or similar) reconfiguration. A simpler approach is to undo the changes done so far and re-establish the existing system. In both cases it means that the existing change specification has to be changed while it is being applied, i.e., the configuration manager which is executing the changes must be reconfigured dynamically.

The other situation concerns the validation process. A change in a subsystem (like the removal without replacement of a server) may force some changes to be done in other parts of the system (like substituting a dependent transaction by an independent one). Thus the validation of the change specification for a subsystem may force the change specifications of other subsystems to be changed. On the other hand, this means that the validation of those systems must be redone which may cause further changes in the specifications of other subsystems and so on. Again, change specifications may change dynamically (in this case as they are validated).

References

[1] M. Endler. A language for implementing generic dynamic reconfigurations of distributed programs. In *Proceedings of the 12th Brazilian Symposium on Computer Networks*, pages 175–187, Curitiba, May 1994.

[2] K. M. Goudarzi and J. Kramer. Maintaining node consistency in the face of dynamic change. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, pages 62–69, Annapolis, MD, USA, May 1996. IEEE Computer Society Press.

[3] C. Hofmeister and J. Purtilo. Dynamic reconfiguration in distributed systems: Adapting software modules for replacement. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 101–110, Pittsburgh, May 1993. IEEE Computer Society Press.

[4] T. Kindberg. Reconfiguring client-server systems. Technical Report QMW-DCS-1993-630, Queen Mary and Westfield College, Department of Computer Science, Mar. 1993.

[5] J. Kramer and J. Magee. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, 11(14):424–435, Apr. 1985.

[6] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, Nov. 1990.

[7] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the Fifth European Software Engineering Conference*, Barcelona, Sept. 1995.

[8] J. Magee, J. Kramer, and M. Sloman. Constructing distributed systems in CONIC. *IEEE Transactions on Software Engineering*, 15(6):663–675, June 1989.

[9] I. Warren and I. Sommerville. Dynamic configuration abstraction. In W. Shafer and P. Botella, editors, *Proceedings of the Fifth European Software Engineering Conference*, number 989 in LNCS, pages 173–190, Sitges, Spain, Sept. 1995. Springer-Verlag.