Synchronisation in Trust Management using Push Authorisation

Thomas B. Quillinan and Simon N. Foley

Department of Computer Science,
University College, Cork, Ireland.
{t.quillinan, s.foley}@cs.ucc.ie

Abstract

Traditional trust management authorisation decisions for distributed technologies, are, in general, based on the history of the authorisations/computation to date. We consider this a *pull authorisation* strategy: the authorisation decision reflects the current and/or past authorisations. In this paper, we examine this pull strategy and propose an alternative form of authorisation in a distributed environment. Instead of 'pulling' the information required for the current authorisation decisions from the past, authorisation decisions are made to specify what will happen in the future. This strategy is called *push authorisation*. When a push decision is made, its result is pushed to just the relevant protection mechanisms. This approach allows the creation of distributed separation of duties policies, without requiring additional synchronisation between components in the execution. It allows present actions to inform future authorisation decisions, before those decisions must be made.

Key words: Naming, Trust Management, Distributed Systems.

1 Introduction

Distributed computing technologies, such as Grid and cluster computing, raise unique problems when articulating security policies. With traditional closed systems, computations are performed within domains, where attributes are a priori known. Distributed applications are made up of computational components that are executed on distributed resources. Traditional authorisation decisions, including trust management [2,5], are, in general, based on the history of the authorisations/computation to date. We consider this *pull authorisation*: the authorisation decision reflects the current and/or past authorisations. For example, in a Chinese Wall policy [3], having previously accessed a particular organisation dataset, a consultant is not permitted access the dataset of a competitor. Other examples of pull authorisation include high water mark mechanisms [16], trace based authorisation [15], and dynamic separation of duties [10]. However, supporting pull authorisation in a distributed environment can be difficult. It requires protection mechanisms to ‘pull’ authorisation state from a variety of sources to ensure a complete view of the authorisation history across the distributed system. In the case of a Chinese Wall policy, before deciding whether a consultant may access a Shell dataset, the protection mechanism must pull past authorisation details from network authorisation servers. For example, in [1], authorisation decisions and are stored within self-describing workflows and are pulled from one task to the next as required.

We challenge this conventional pull strategy and propose an alternative perspective of authorisation in a distributed environment. Rather than current authorisation decisions being ‘pulled’ from past authorisation history, authorisation decisions prescribe what will happen in future authorisations. This strategy is called *push authorisation*. Instead of authorisation state having to be pulled from all authorisation sources, when a push authorisation decision is made, its result is pushed to just the relevant protection mechanisms. As will be demonstrated in this paper, this subtle difference in perspective considerably simplifies the architectural support that is required to coordinate the distributed authorisation state.

In this paper we propose a distributed computational model than supports pull and push authorisation and demonstrate the effectiveness of the push strategy. We extend the naming architecture described in [11] and show how it can be used to provide support for both of these authorisation strategies

This paper is organised as follows. Our research uses the Condensed Graphs [8] model of distributed Computation and this is outlined in Section 2. Section 3 describes a naming system that extends the scheme in [11] to support push authorisation. Section 4 describes pull authorisation within this model. In Section 5, we introduce push authorisation and provide examples demonstrating it use.

2 Condensed Graphs

Previous work has examined the use of trust management in providing a security architecture within the WebCom metacomputing environment [6]. WebCom supports heterogeneous applications, using components based on technologies such as Grid and Web Services. In general, applications are made up of distinct functional nodes. Each node takes input, operates on it and returns a result. In order that compute resources can make decisions regarding these functional nodes, security checks are required based on the input data; the function of the node; the result and the overall purpose of the application. In addition the user launching these applications must be confidant in the compute resources being utilised by the computation.

In the WebCom System, applications are coded as hierarchical condensed graphs [8] which provide a simple notation in which lazy, eager and imperative computation can be naturally expressed. There are two types of distributable operation: nodes that represent atomic operations and condensed nodes that represent subtasks and are encapsulated as subgraphs. Atomic operations are value-transforming actions and can be defined at any level of granularity, ranging from low-level machine instructions to mobile-code programs such as applets, middleware components (such as .NET, Globus).

Example 1 The Condensed graph shown in Figure 1 defines a simple Web Services application. This application is defined as a workflow of atomic actions. The

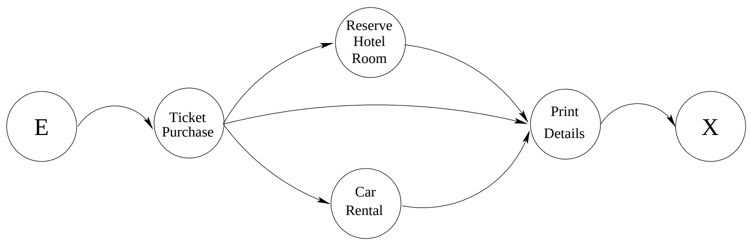
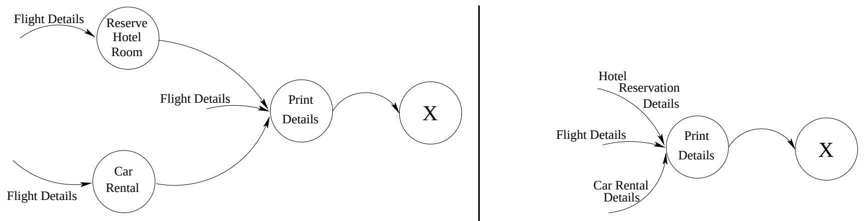


Fig. 1. A simple Travel Agent Web Services application, specifi ed as a Condensed Graph.

application operates as a simple travel agent, using web services from different sites. Users of the application are directed to fill in the details required to purchase an airline ticket. Once this purchase is completed, the relevant details are sent to hotel reservation and car rental sites. The user can then fill in any extra details. Finally all the details are collated and printed out for the user. The graph specifies the sequencing constraints of the application components. Nodes perform actions, and data travels along the arcs between the nodes. Nodes can be thought of as the meeting place for the execution triple: the inputs to the node, the operator and the destination. Nodes execute when this triple is complete. Figure 2 shows the graph from Figure 1 in the process of firing. As nodes fire and results are returned, more nodes become fireable.



(a) Graph after a plane ticket has been reserved.

(b) Graph after the plane, car and hotel reservations.

Fig. 2. Firing sequence of the Web Services Application Graph.

Any node in this graph could itself be a Condensed Graph. Furthermore this graph, called `TravelAgent`, could itself be part of another Condensed Graph. Recursive graphs are also possible, for example, the TravelAgent graph could include an instance of itself. △

As the application graph executes, nodes become "fireable" (the execution triple is complete). When this occurs, the node is scheduled for execution to some execution context. When a node executes, its result is returned and integrated into the graph, making more nodes fireable. Thus as the execution progresses, the graph is modified to represent the current state of the application.

2.1 Authorisation in Condensed Graphs

WebCom [9] is a distributed architecture for executing, and for coordinating the execution of, Condensed Graphs. WebCom Masters use trust management credentials [2,13] to determine the operations that the client is authorised to execute; WebCom master credentials are used by clients to determine if the master had the authorisation to schedule the computation that the client is about execute.

In the original Secure WebCom [6], authorisation decisions were made based only on the condensed node function, such as `CarRental`. The policies that can be defined are limited by the amount of information available to the security infrastructure, in effect the trust decisions that can be made are coarse grained. In [11], we extend the WebCom authorisation scheme to include additional information about the other attributes of a component. In this paper we build on this extended WebCom authorisation model to support richer authorisation policies in order to explore the notions of pull and push authorisation.

3 Naming

Authorisation in Secure WebCom requires the ability to precisely refer to the components of concern in relevant ways. For example, one may wish to be able to distinguish between a `CarRental` node that executes using credit card details rather than PayPal. We argue that this problem can be reduced to a *naming* problem. Naming distributed components [7,12] is typically static; for example, names used in Web Services [4]. Naming in WebCom is dynamic in order to reflect the dynamically evolving nature of the nodes in a condensed graph.

3.1 Naming Computations

Dynamically generating a name in a computation requires capturing the attributes of the computation at a specific moment. We can use these names to make trust decisions regarding the ongoing computation. To properly name a component in a computation, we must first identify the computational context that identify that component. A computation is comprised of a collection of *execution contexts*, each one corresponding to the nodes in the executing condensed graph.

Definition 3.1 *An Execution Context, expressed as S-Expression [14], is a 5-tuple:*

- *Domain: the name of execution domain to which this component is currently executing, or is scheduled to execute.*
- *Application: the name of the condensed graph in which this component appears.*
- *Function: the name of the operational function of this component.*
- *Inputs: the names of the inputs to this node.*
- *Outputs: the names of the destination nodes for output.*

An execution context is used to represent a computation in various states of completion. As a consequence, some of its attributes may be null, reflecting, for example, currently unavailable input values, destinations, and so forth. A *name* may, in turn, also refer to further execution contexts. Avoiding circular definitions is achieved through reduction rules considered in Section 3.3.

3.2 Naming Condensed Graphs

SDSI-like local naming is used to name the attributes of an execution context. Principals in SDSI [13] define their objects according to their local view of the system. Local naming provides the ability to use names from other principals namespaces. Applying this principle to Condensed Graphs, we can define, for example, the input to a node as *"Node's Input"*, from the perspective of the graph. Expanding this name further we get *"Graph's Node's Input"*. Using this approach, we name every portion of the graph in as much detail as is required to uniquely identify it. For example, from Alice's perspective, the name of the `CarRental` node from Figure 1 executing on her computer can be specified as shown in Figure 3.

```
((Computer's TravelAgent),
 (Computer),
 (Computer's TravelAgent's CarRental),
 (Computer's TravelAgent's CarRental's Input),
 (Computer's TravelAgent's CarRental's Output))
```

Fig. 3. The Name of the `CarRental` node from Alice's perspective

From Bob's perspective, the name must refer to Alice, in whose namespace these name-components exist. Local naming provides the ability to store the required detail to uniquely identify each portion of the nodes in as much detail as is necessary. These names are primarily used within WebCom [11] and thus each computational context is also given as a WebCom name.

Figure 4 defines the syntax of a WebCom name. All parts of the name are optional, a name can be represented by a combination of any of these fields or by a simple S-Expression. There can be one or more `input` and/or `output` fields when the `inputs` and/or `outputs` fields, respectively, are present.

```
<webcomname> ::=
  (WebComName
    [(domain <webcomname>)] [(graph <webcomname>)]
    [(function <webcomname>)]
    [(inputs  (input <webcomname>)  {(input <webcomname>)}  )]
    [(outputs (output <webcomname>) {(output <webcomname>)} )]
  )
<webcomname> ::=
  (WebComName S-Expression)
```

Fig. 4. The format of a generic WebCom Name.

Example 2 Figure 5 gives a WebCom name for the node `CarRental` from the Condensed Graph in Figure 1. Alternatively, a representation of this node could include less information, such as the simple S-Expression:

```
(WebComName (ref: RentalCompany CarRental))
```

```
(WebComName
  (domain (ref: RentalCompany (ref: Location)))
  (graph (ref: Company (ref: Office (ref: Alice TravelAgent))))
  (function (ref: RentalCompany (ref: Location
                                (ref: TravelAgent CarRental))))
  (inputs (input (ref: Airline (ref: Flight (ref: Destination
                          (ref: TravelAgent TicketPurchase))))))
  (outputs (output (ref: Company (ref: Office (ref: Alice
                            (ref: TravelAgent PrintDetails))))))  )
```

Fig. 5. An S-Expression version of the name for the `CarRental` node.

representing a node that Alice refers to as "Rental Company's CarRental". In Bob's namespace, this is "Alice's RentalCompany's CarRental". △

While these names provide the contextual detail required to enable authorisation policies to be articulated before computation takes place, it is clear that the size of these names will cause them to become unusable in computations of a non-trivial nature. A system is required to provide a canonical form for these names, yet still containing enough detail to allow informed authorisation decisions to be made.

3.3 Reduction Rules

The contextual detail provided by WebCom names comes at the cost of possible redundant information stored in the name. This cost can be reduced through the use of *reduction rules*. A reduction rule is a heuristic by which an abstract name can be translated into a more compact, yet equivalent form. Having a unique reference to an node is not always ideal. Creating security policies may require generic node references. For example, it is not always appropriate to make security decisions based on the path that an execution has taken to this point. This would require knowledge of all valid paths that the computation would be allowed to take. Reduction rules can be used to create more generic names for use within policies.

In [11], we examined some basic reduction rules. These rules considered tuple elimination rules only, for example removing Domains or Inputs from the name to make them more usable. However, such rules are very limited. We need more sophisticated reduction rules that consider the contextual state of the execution. These rules must retain details important to the security policy of the system, while eliminating irrelevant details. They must create a canonical form that can be used to write and enforce authorisation policies.

3.3.1 Tuple Reduction

Reducing the components of a name is an application specific process, as, for example, the type of the inputs and results to nodes are potentially unique to an application. Each application defines the specific rules for that application. For example, a rule for the graph shown in Figure 1 could specify that the flight details returned by the *TicketPurchase* node is reduced to contain the airline name, destination and travel date, as these may be important for the security policy. This reduction makes no change to the results returned by a node, just how they are represented to the security infrastructure. We will examine these rules in more detail in Sections 4 and 5, in the context of specific examples.

3.3.2 Tuple Elimination

The simplest form of reduction rule is a tuple elimination rule. With these rules, component tuples, for example the domain tuple, of the name are removed. We represent reduction rules with a simple logic axiom, `reducesTo(X,Y)`. This means that where the pattern Y is found, it is replaced with X. Figure 6 defines the five basic tuple elimination rules. Each axiom replaces a component tuple with a null s-expression (∅). These rules can be used individually, or in combination.

```
reducesTo(name(∅,g,f,i,o), name(d,g,f,i,o)).
reducesTo(name(d,∅,f,i,o), name(d,g,f,i,o)).
reducesTo(name(d,g,∅,i,o), name(d,g,f,i,o)).
reducesTo(name(d,g,f,∅,o), name(d,g,f,i,o)).
reducesTo(name(d,g,f,i,∅), name(d,g,f,i,o)).
```

Fig. 6. Reduction rules used to remove component tuples

4 Pull Authorisation

Traditional authorisation decisions are made based on the context in which components have been executed in the past. With this “pull authorisation” strategy, the authorisation state needed to make decisions must be pulled from all the distributed mechanisms that are involved in the computation. In the case of trust management, this means that all credentials issued must be pulled to accurately determine the authorisation history of a user.

Definition 4.1 *Pull Reduction: A node n has a name (execution context)*

$$[D, G, F, [i_1, i_2, \ldots, i_n], O]$$

whose attributes provide the names of execution contexts for domain (D), application graph (G), function (F), inputs ($i_1, \ldots, i_n$) and outputs (O), respectively. The pull reduction of the name of node n is

$$\mathcal{E}[D, G, F, \mathcal{R}[i_1, i_2, \ldots, i_n], O]$$

where $\mathcal{E}[\ldots]$ and $\mathcal{R}[\ldots]$ represent the application of tuple elimination and application reduction rules, respectively

To perform a ‘pull’ authorisation, we take the inputs to a node and examine their execution contexts. This takes the form of a sequence of reduction rules. First, the input tuple is examined and its context is acquired. Figure 7 shows a representation of such a rule in the form of a logic axiom. This axiom defines that

```
reducesTo(name(d,g,f,name(d′,g′,f′,i′,o′),o), name(d,g,f,i,o)).
```

Fig. 7. Input Expansion Rule

an execution context is expanded to contain the context(s) of the input(s). Next the relevant details from the input tuple are extracted and are integrated into the name of the current context. This is the $\mathcal{R}$ reduction rule. For example, if there existed an ordering where the graph of the input were greater than the graph of the current context, the graph of the input's context would replace that of the current graph. Example rules to achieve this are shown in Figure 8.

In this case the `order(g′,g)` axiom states that there exists an ordering g' $\geq$ g, such that the graph g' takes precedence over g. Therefore the current context is modified to contain the input's graph name.

```
reducesTo(name(d,g′,f,i,o),name(d,g,f, name(d′,g′,f′,i′,o′),o).
order(g′,g).
```

Fig. 8. Domain ordering rules

Tuple elimination rules, such as those discussed in Section 3.3.2, are then applied to reduce the name to the form required by the authorisation mechanism. These act as the $\mathcal{E}$ reduction rules. An authorisation decision can then be performed based on the node name. This mechanism allows the enforcement of history-based, or pull, authorisations within, for example, a trust management system. The node names form the attributes used when making a trust decision and credentials holding these names are delegated to the principals involved in the computation.

The pull authorisation strategy works well in the case of a straight ordering of, in this case, domains. However, consider the case where we have a mutually exclusive ordering such as with a separation of duties policy. With a Condensed Graph application, it is possible that multiple nodes execute in parallel, and the results from these parallel executions are integrated in the future. If we use a pull strategy to enforce a separation of concerns policy, we can encounter deadlock, where the computation can never finish due to a policy conflict. We could address this concern with a policy that dictates all computations must execute in one domain, however this will force the computation to be assigned a priori to a specific domain. This approach limits the flexibility of the computation, Instead we introduce the concept of 'push' authorisation to address this issue.

Example 3 In a high watermark policy component names rise to reflect the classification of the data written to it. In the case of a Condensed Graph application, this is cast as a policy where a node may only execute on a resource of equal or higher classification. The classification of a node will depend on the path the execution has taken to this point. In a distributed computation such a policy may be expressed as: "once a computation is executed on a resource running at a certain security level, resources that are of a lower security level should never be used in the future execution of the computation". Traditionally, such policies required a centralised method to store state. An infrastructure is used to maintain the contexts that the computation has used to that point.

We use node names to store security state in a decentralised manner during application execution. The details required to make authorisation decisions are pulled from the nodes that have executed. The reduction policies applied to the names must encode the highest security clearance of the resources that the execution has used to this point. In the case of a distributed computation, the resources that are used to execute components in the future depends on the resources used to date.

Figure 9 shows a Condensed Graph that describes how an airline implements a flight reservation application. This graph can be considered as an implementation of the `TicketPurchase` node from Figure 1. The airline's policy includes a

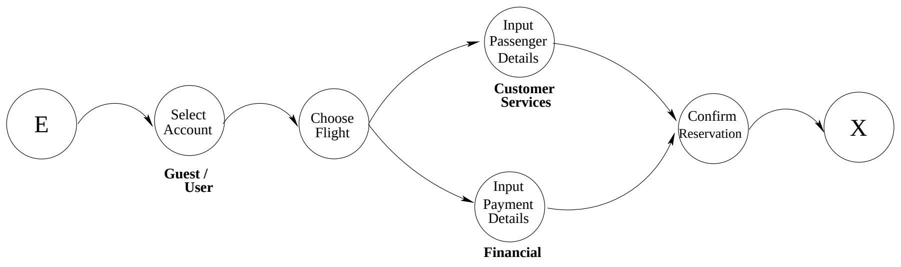


Fig. 9. Reserving a Flight specifi ed as a Condensed Graph.

basic ordering of domains, Financial ≥ CustomerService ≥ Guest. For example, if a node executes on a resource that is classified *Financial*, no subsequent node should execute on resources classified *CustomerService* or lower.

If this graph executes on the Airline's network, and in the domains shown, the security policy should require, that the `ConfirmReservation` node should only be executed in the *Finance* domain. This can be achieved using reduction rules that maintain the high watermark within a node's name.

We use the high watermark reduction rule shown in Figure 10. This operates in a twofold manner. First, the ordered pair, `order(d',d)`, indicates that d' ≥ d. This means that wherever d is present, d' replaces it. Second, the input context is examined and if the input's domain tuple matches the ordering, it replaces the domain in the current context.

```
reducesTo(name(d',g,f,i,o), name(d,g,f,i,o)) :-
  order(d',d), name(d,g,f,name(d',g',f',i',o'),o).
```

Fig. 10. High watermark reduction rule.

In the names shown in Figure 11, prior to the application of the reduction rules, the node `ConfirmFlight` is considered to be permitted to execute on a resource classified *CustomerService*. However, as one of the previously executed nodes, `InputPaymentDetails`, was executed on a resource of classification *Finance*, the Reduction rule modifies the name of the node and specifies that it must execute on a resource of at least *Finance*.

This enforces a high water mark authorisation policy, once any node in the computation reaches a higher security level, all subsequent nodes must execute on resources with at least that security level. We could also make decisions based on the type of customer using the service. If a known customer, whose details are retained by the airline, logs into the site, the name of the node could be set to reflect

```
(WebComName
  (domain CustomerService)
  (graph TicketPurchase)
  (function ConfirmFlight)
  (inputs (input (WebComName (domain CustomerService)
                             (function InputPassengerDetails)))
          (input (WebComName (domain Finance)
                             (function InputPaymentDetails))))
  (outputs (output X)) )
```

(a) Before Reduction

```
(WebComName (domain Finance) (function ConfirmFlight))
```

(b) After Reduction

```
reducesTo(name(d,∅,f,∅,∅), name(d,g,f,i,o)).
```

(c) The Tuple Elimination Reduction Rule

Fig. 11. The name of the `ConfirmFlight` node, (a) before and (b) after reduction, and (c) the reduction rule used.

this. When the payment details are required, that node would be directed to specific resources that hold saved customer financial details. △

5 Push Authorisation

Authorisation decisions often have consequences that alter the possible future authorisation decisions that may occur. For example, in a Chinese Wall policy [3], once a computation executes on a particular company's resource, it should never be allowed execute on resources belonging to a competitor. One solution to this issue is to select *a priori* where the computation will execute. However, this limits the flexibility of the system to adapt to changes at runtime.

Traditionally, when such problems are addressed dynamically, they use synchronisation between the components. However, such synchronisation is often undesirable and requires extra infrastructural support. Ideally, we want to be able to identify potential conflicts and address them within the authorisation policy. Instead of pulling the information required to make a decision from the source, we instead *push* this information from the source to the points where it will be needed. We refer to these as 'push' authorisation policies.

We can use push authorisation to force a computation to execute in the future on specific resources based on the authorisations that the computation has received in the past and on the potential conflicts that must be avoided in the future. This allows, for example, the enforcement of dynamic separation of duty policies, without an external synchronisation infrastructure. The details required to identify potential conflicts are stored in the computational context of the nodes. Push authorisation policies are then written in terms of trust management credentials and are enforced using the existing security architecture. No additional architecture is required to support these policies, the only change is in how the names of the node are created.

Push reduction can be defined in a similar way to pull reduction. The subtle difference between them exists in the part of the execution context that is expanded, inputs for pull reduction, and outputs for push reduction.

Definition 5.1 *Push Reduction: A node n has a name (execution context)*

$$[D, G, F, I, [o_1, o_2, \ldots, o_n]]$$

whose attributes provide the names of execution contexts for domain (D), application graph (G), function (F), inputs (I) and outputs ($o_1, \ldots, o_n$) , respectively. The push reduction of the name of node n is

$$\mathcal{E}[D, G, F, I, \mathcal{R}[o_1, o_2, \ldots, o_n]]$$

where $\mathcal{E}[\ldots]$ and $\mathcal{R}[\ldots]$ represent the application of tuple elimination and application reduction rules, respectively

To make a 'push' authorisation decision about an execution context, first we expand the execution context to examine the destination contexts that will be used. Next application specific tuple reduction rules, $\mathcal{R}$ are used to extract the relevant details. Tuple elimination rules, $\mathcal{E}$ are then used, if required. Finally the reduced name is used with the authorisation mechanism.

Example 4 The rental company's separation of duties policy states that: "Sales and Finance Data should not be processed on the same resource". Once the node

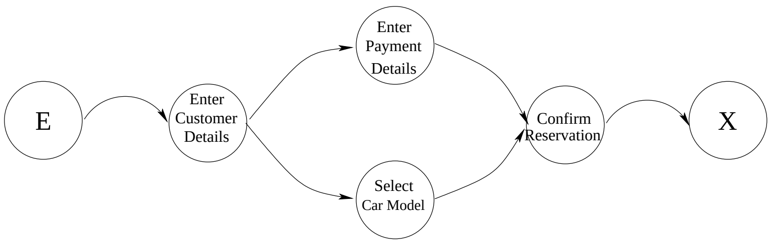


Fig. 12. Reserving a Car specifi ed as a Condensed Graph.

`EnterPaymentDetails`, from Figure 12, has been scheduled to a computational context, the name of `SelectCarModel` node must be updated so that when it is to be scheduled, it is sent to a non conflicting domain. This ensures that when the results from both of these nodes reaches the `ConfirmReservation` node, it can be executed in accordance with the separation of duties policy.

Updating the `ConfirmReservation` node could be achieved with pull authorisation; however updating the `SelectCarModel` node would require communication between nodes that are able to execute at the same time to ensure synchronisation. Instead, when the `EnterCustomerDetails` node's result is to be integrated into the computation, the details of the computational contexts that it will be sent to is pushed to the `SelectCarModel` node.

Figure 13 shows a representation of the node's name before a push authorisation decision has taken place. In particular, the node's domain has not been specified at this point. When the `EnterPaymentDetails` node is scheduled to execute, the authorisation policy ensures it is sent to the *Finance* domain. Thus, all subsequent

```
(WebComName (graph CarRental) (function SelectCarModel)
  (inputs (input(WebComName (domain CustomerService)
                            (function EnterCustomerDetails)))
  (outputs (output (WebComName (function ConfirmReservation)))) )
```

Fig. 13. Name of the `SelectCarModel` node before the Push Authorisation decision.

nodes must adhere to the separation of duties policies. Once this requirement is apparent, the name of the `SelectCarModel` node is modified to contain a specific domain, *Finance*. This push action ensures that no policy conflicts will occur.

```
reducesTo(
  name(d,g,f,i,(name(d*,g',f',i',o'),name(d*,g*,f*,i*,o*))),
  name(d,g,f,i,(name(d',g',f',i',o'),name(d*,g*,f*,i*,o*)))).
```

Fig. 14. Push reduction rule, for a node with two destination domains, d' and d*.

This push action is implemented in the form of a reduction rule. Figure 14 shows such a rule where a node's result may be sent to conflicting domains `d'` and `d*`. The `reducesTo` axiom defines that where the result of this node could execute in domains `d'`and `d*`, then they should both be forced to be executed in domain `d*`. These rules are applied to the names of the nodes that will execute in the future and the authorisation mechanism ensures that they are scheduled to the correct domains. Figure 15 shows the node name after the push. △

```
(WebComName
  (domain Finance) (graph CarRental) (function SelectCarModel)
  (inputs  (input (WebComName (domain CustomerService)
                              (function EnterCustomerDetails))))
  (outputs (output (WebComName (function ConfirmReservation)))) )
```

Fig. 15. Name of the *SelectModel* node after the authorisation decision.

Push authorisation allows a more dynamic control over ongoing computations and provides support for pushing computations to specific resources using the security policy. This allows the implementation of distributed separation of duty policies, without requiring ongoing synchronisation or communication between atomic nodes. Changing the names of the nodes requires no changes to the existing pull-based security architecture. The same security policies are used to provide authorisation. The only change is in how the relevant information is provided to the protection mechanism. This subtle change is powerful. Traditionally, synchronisation between parallel processes requires additional hooks into the application.

With this push mechanism, we can provide the communication within the existing framework. However, providing a push authorisation model limits the possible contexts that a computation may execute in the future. The fact that decisions are pushed before execution means that some potential future information cannot be used when making a decision. We argue that the advantages that simplification of the architecture bring, outweigh this potential downside.

Push authorisations can be modelled within the pull architecture, however this requires a centralised synchronisation mechanism. When a pull reduction takes place, the node names on parallel paths must be synchronised. Such a centralised mechanism would have to exist outside of the existing trust architecture.

6 Implementation

Integrating the naming architecture into the WebCom architecture is achieved through the creation of a security manager module to support it. This security manager takes a node, extracts it's *WebCom name* and reduces it according to the reduction rules of the particular application. This name forms part of the query to a trust management system, along with the system policy and the appropriate client's credentials. If the trust management system finds a trusted path from the system policy to the client's key, it notifies the security manager. The security manager in turn notifies the scheduler that a suitable client has been found. When a result is returned from a client, before it is integrated into the execution, a security check is performed to ensure the system policy has been upheld.

7 Discussion and Conclusion

Traditional authorisation policies use a history based mechanism to determine if current actions are authorised. With this *pull authorisation* strategy, the information required to make an authorisation decision is pulled from source. However, supporting pull authorisation in a distributed environment can be difficult. It requires protection mechanisms to 'pull' authorisation state from a variety of sources to ensure a complete view of the authorisation history across the distributed system.

In this paper we propose an alternative approach. Rather than pulling authorisation decisions from past history, authorisation decisions prescribe what will happen in future authorisations. We call this strategy *push authorisation*. Instead of pulling authorisation state from all authorisation sources, when a push authorisation decision is made, its result is pushed to just the relevant protection mechanisms.

This subtle difference in perspective considerably simplifies the architectural support that is required to coordinate the distributed authorisation state. Using the naming architecture developed for Condensed Graphs, we can enforce both pull and push authorisation mechanisms. Providing support for the push strategy allows development of authorisation policies without requiring communication between separate paths in the future. Using the naming architecture of the Condensed Graphs environment, we can support this push strategy. As only the names of the components change, no modification of the authorisation system is required.

Acknowledgements

This work was supported by the Boole Centre for Research in Informatics, University College Cork under the HEA-PRTLI scheme and by the Enterprise Ireland Commercialisation grant Scheme.

References

[1] V. Atluri, S.A. Chun, and P. Mazzoleni. A chinese wall security model for decentralized workflow systems. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 48–57, 2001.

[2] M Blaze et al. The keynote trust-management system version 2. September 1999. Internet Request For Comments 2704.

[3] D.F.C. Brewer and M.J. Nash. The Chinese Wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214, May 1989.

[4] Distributed Management Task Force, http://www.dmtf.org/standards/cim/cim_schema_v291_prelim. *Common Information Model (CIM) Version 2.9.1 Specification*, 21 February 2005.

[5] C Ellison et al. SPKI certifi cate theory. September 1999. Internet Request for Comments: 2693.

[6] S.N. Foley, T.B. Quillinan, and J.P. Morrison. Secure component distribution using webcom. In *Proc. 17th IFIP Int. Conf. on Information Security*, Cairo, 2002.

[7] Internet Engineering Task Force. Public key infrastructure (x.509) [PKIX]. http://www.ietf.org/html.charters/pkix-charter.html.

[8] J.P. Morrison. *Condensed Graphs: Unifying Availability-Driven, Coercion-Driven and Control-Driven Computing*. PhD thesis, Eindhoven, 1996.

[9] J.P. Morrison et al. WebCom-G: Grid enabled metacomputing. *Neural, Scientific and Parallel Computations Journal.*, 2004.

[10] M.J. Nash and K.R. Poland. Some conundrums concerning separation of duty. In *Proceedings of the Symposium on Security and Privacy*, pages 201–207, 1990.

[11] Thomas B. Quillinan and Simon N. Foley. Security in WebCom: Addressing naming issues for a web services architecture. In *Proceedings of the 2004 ACM Workshop on Secure Web Services (SWS).*, Washington D.C., USA., October 2004. ACM.

[12] Sanjay Radia. Naming policies in the spring system. In *Proceedings of the 1st International Workshop on Services in Distributed and Networked Environments*. Sun Microsystems, Inc., IEEE, 1994.

[13] R Rivest and B Lampson. SDSI - a simple distributed security infrastructure. In *DIMACS Workshop on Trust Management in Networks*, 1996.

[14] Ronald L. Rivest. S-expressions. Technical report, Network Working Group, May 1997. Internet Draft: http://theory.lcs.mit.edu/ rivest/sexp.txt.

[15] A.E.K. Sobel and J. Alves-Foss. A trace-based model of the chinese wall security policy. In *Proceedings of the 22nd National Information Systems Security Conference*, Arlington, Va., USA, October 1999.

[16] C. Weissman. Security controls in the ADEPT-50 time-sharing system. In *AFIPS Conference Proceedings*, volume 35, pages 119–133. FJCC, 1969.