Computer Science and Information Systems 11(1):291–308 DOI: 10.2298/CSIS130204009Z

Duplication Problem in Treaty Systems: Causes and Solutions

Yining Zhao[1] and Alan Wood[2]

[1] Department of Computer Science
University of York, York, UK
yz616@york.ac.uk
[2] Department of Computer Science
University of York, York, UK
alan.wood@york.ac.uk

Abstract. Capabilities are a more scalable and adaptive access control approach compared with the conventional approaches such as ACLs, due to their being held and managed by users or agents in systems, but not the middleware. This feature makes capabilities more suitable in distributed environments that have dynamic populations. Treaties have been proposed to enhance the capability approach by introducing sequences of actions, such that treaties can capture characteristics of behaviours, and provide finer control over accesses. However there is a new problem brought by the behaviour modeling of treaties which is called *duplication problem*, which concerns preventing users from gaining unauthorized behaviour by duplicating treaties. In this paper we provide the formal definitions of treaty operations, and discuss the causes of the duplication problem, and how treaty operations can affect this. We also propose three models of treaty systems that aim to solve the duplication problem, and evaluating their performance and scalability.

Keywords: Behaviour Control, Access Control, Duplication Problem, Treaties, Distributed Computing.

1. Introduction

Distributed systems and cloud services are important areas of research that are becoming ubiquitous components in everyone's life. It is essential to provide these applications with the necessary support mechanisms that ensure functionality, confidentiality, integrity, security, etc. Most modern services are provided subject to some required protocols which can be expressed as allowable sequences of actions. For example, to use an on-line banking system, the user would need to first login, and then choose to do a number of actions such as making payments or transferring money between accounts, and then logout. Clearly these services involve confidential data that should be protected and hence access (and other) control mechanisms are essential.

In order that such services can be provided, the system must provide mechanisms to support:

- confidentiality and security — only authorised agents can access certain resources;
- agent-level construction of behavioural specifications;
- flexibility and scalability, so that they are able to operate in open distributed environments.

Most conventional access control mechanisms are centralized, since agents' identities (and other properties) need to be managed centrally. Although this option makes certain useful properties easy to implement, such as strong authentication, these mechanisms are quite static and scale badly.

Capabilities, originally introduced by Dennis and Van Horn [1] to support memory management for multi-programmed computation, are a well-known dynamic approach to access control. They differ from the widely used Access Control List (ACL) approach, as ACLs are centralized controls with a fixed user population, whereas capabilities are more flexible and can be easily distributed, due to that they can be propagated among users.

Capabilities also have some weaknesses that may cause problems. For example, it is widely believed that capabilities cannot enforce confinement [8], nor to revoke permissions that have been previously granted, as being addressed the revocation problem [7][4]. People have worked with these and suggested some solutions (see section 2).

In addition to these apparent problems, the structure of capabilities is not rich enough for defining agents' behaviours in distributed systems. For instance, the holder of a capability has unlimited use of any right that the capability provides. Capabilities cannot express cases such as 'only allowed to read the file three times'. These and other considerations mean that we need to develop new solutions that may better support services which require a finer level of behavioural specification.

Here we introduce treaties as a new approach to access control in distributed environments. In treaties there are behaviour descriptors that are used for modelling the sequences, branches and terminations of actions. This distinguishes treaties from capabilities that contain 'sets of rights'. With the new component, treaties can define the accesses of users in a much more detailed and accurate way.

In section 3 the concept of treaties will be reviewed, with the duplication problem being detailed in the following part, introducing three models of treaty systems that aim to solve the duplication problem. Treaties in these models have different locations which may affect the structure and efficiency of treaty systems. Hence we will also examine the performance of these models in the evaluation section.

2. Related Work

There has been much work aimed at the improvement of the capability approach to overcome the restrictions mentioned above and make it more applicable. A known mechanism that solves the revocation problem is the use of caretaker [9]. When a user passes capabilities to others, he can add 'caretakers' to them. The caretakers refer to some 'gates' set up by the user, and the user can 'open' or 'close' these gates. Only when gates are open, can others use capabilities from the user to access resources. Therefore the user can disable capabilities previously sent out by shutting down gates, i.e. he revokes these capabilities. ICAP [4] and SplitCap [7] are claimed to solve confinement problems to some extent.

Capabilities with modifications are also applied in various fields. EROS [10] shows that a capability-based operating system can perform as well as ACL-based systems without any special hardware assists. Establishing private channels in LINDA-like [3] tuple-space systems has also been suggested using mix of ACLs and capabilities [12]. Multicapabilities [11] apply the idea of capabilities into tuple-space systems by modifying the structure of capabilities to refer to multiple unnamed targets of a matching pattern. μKLAIM [5] uses a capability approach to solve the problem of unmatched static control in dynamic environments. Vistas [13] extend the idea of capabilities by introducing combining operations, so that the holder of vistas can construct them to fit different cases. Vistas provide a 'product' operation which displays some aspects of behavioural specification by constructing a pair of actions that must be performed in sequence. Vistas can also contain references to multiple objects which capabilities do not. The treaty concept is a direct extension of the idea of vistas, and they can be represented in various options [14].

Session Types [2] provide a similar fashion of behaviour modelling that treaties are capturing. Session types are used to define communications with a prescribed conversation plan, so that the types and sequences of messages can be expected. In that way they can ensure the two communicating parties are using the same and correct protocol, and the communication patterns and flow controls are correctly implemented. However, the orientation of session type theories is different from treaties, as session types mainly concern the verification of communications at compile-time, while treaties are making their effects at run-time.

3. The Concept of Treaties

Treaties are extended from the idea of capabilities, and they can also be used as access control components. This inheritance implies that treaties adopt many properties from capabilities. They are also unforgeable pieces and can be propagated, thus suitable for distributed environments.

The key function that treaties provide is 'behaviour control'. This differs from the 'rights only' mode of capabilities in the sense that a treaty can allow or deny different actions at different times or stages, depending on the current state of the treaty or current stage of the process/behaviour (capabilities are stateless entities, and so cannot control sequences of actions. See Fig. 1). With treaties sequences of allowed actions are specified, and enforced, in order to form behaviours. Treaties aim to capture attributes of behaviours, such as sequencing, branching and terminating. This is done by introducing 'states' into treaties.

Abstractly, treaties are access control components that provide specifications of behaviours that can be followed for a number of resources, together with the current state of the evolution of the behaviours. As fundamental requirements, treaties shall:

- refer to resources.
- provide information to a kernel to enable it to allow or deny an action request.
- not be strictly bound to agents, i.e. kernels do not need to check their holder's identity in order to grant or deny actions.
- be able to 'evolve' behaviours according to their specifications, i.e. maintain and update the behavioural state.

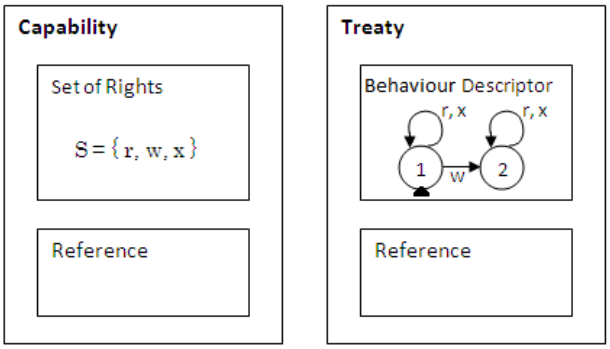

Fig. 1. Structure of Capabilities and Treaties

Therefore each treaty can be seen as consisting of three components: the reference pointing to the resource (or resources), the behaviour descriptor modelling the behaviours permitted by the treaty, and the current state of the behaviour in this treaty. From this, we can see that that capabilities are special treaties, in which there is only a single behavioural state, and all allowed actions are reflexive transitions on this state. We call a capability-like treaty a 'complete treaty', which is created by the kernel when the resource is created, and given to the creator of the resource.

In a treaty-based system, the usual process of using treaties can be like this: a) a user U requires the kernel to create a resource, b) the kernel creates the resource R and returns a complete treaty T to user U, containing all possible action types relevant to the resource R with an unrestricted number of accessing repetitions, c) user U can refine the behaviour model represented by the behaviour descriptor in T so that the specified treaties $T_1...T_n$ are constructed, d) U can pass these specified treaties to other users $U_1...U_n$ to distribute the right to access R, e) U_m who received T_m can show this treaty to the kernel to access R, if this access is validated by T_m.

There are a number of operations that are proposed to refine behaviour models in treaties, and they are called *restrict, intersect, join, concatenate, difference* [14]. The fundamental rule of a treaty system is that an agent cannot increase the totality of behaviours defined by the treaties it holds, hence we need to make sure the given five treaty operations and other potential operations will not break this rule.

The *behaviour set* concept has been discussed in [15]. Briefly speaking, the behaviour set of a treaty α is denoted by $[\![\alpha]\!]$, and it means the set that contain all behaviours that are granted by α, including the empty behaviour ϵ.

We could produce the formal definition of the five basic treaty operations using the concept of behaviour set ($a@b$ denotes the number of times that action a appears in behaviour b):

join	$\alpha \sqcup \beta$	$[\![\alpha \sqcup \beta]\!] = [\![\alpha]\!] \cup [\![\beta]\!]$
intersect	$\alpha \sqcap \beta$	$[\![\alpha \sqcap \beta]\!] = [\![\alpha]\!] \cap [\![\beta]\!]$
difference	$\alpha - \beta$	$[\![\alpha - \beta]\!] = [\![\alpha]\!] \setminus \{ x \mid x = b.\Sigma^*$, where $b \in [\![\beta]\!]$ and $b \neq \epsilon \}$
concatenate	$\alpha \cdot \beta$	$[\![\alpha \cdot \beta]\!] = \{ x \mid x = b_1.b_2$, where $b_1 \in [\![\alpha]\!]$ and $b_2 \in [\![\beta]\!] \}$
restrict	$[\alpha]_n^a$	$[\![[\alpha]_n^a]\!] = \{ x \mid x \in [\![\alpha]\!]$ and $a@x \leq n \}$

These can be used to examine the correctness of the any implementations for the treaty operations, and they are also used to show why using treaty operations without security concerns would cause problems in the next section.

4. The Duplication Problem

Treaties extend vistas, and thus capabilities, in functionality, and so there arise some new practical and theoretical issues that need to be considered. It must also be remembered that treaties not only inherit positive attributes from capabilities but also some negative issues. Thus the development of treaty systems needs to consider these problems, together with their particular challenges brought by treaties.

A main challenge that results from the new functionality of treaties is the *duplication problem*. Treaties require state descriptors to keep track of the evolution of behaviours. This means protecting the state information is crucial. According to the rule of treaties, we must ensure that allowed behaviours cannot be increased. A rollback of current state may destroy this in treaties that are access time sensitive, as it means more actions can be performed other than the assigned actions.

For instance, assume that a user is holding a treaty which allows him to perform a `read` action exactly once. Then the challenge is how to prevent him from validly performing more than one `read` by making a duplicate of the treaty. If this treaty is a piece of data — a bit-string — completely stored in the agent's memory, there is no way of preventing the holder from making a copy of it before using it for any accesses. The holder could then access the targeted resource using the copied treaty, and keep the 'unused' original so that he could do the same thing next time he wants to do a `read` on the resource. In this way this user would gain unlimited number of accesses, which clearly breaks the rule. We address this issue as the duplication problem.

The duplication problem does not only arise in this way. Taking another example of the same treaty, assume the holder A sends (a copy of) his treaty to another user B, and then A uses his copy of the treaty to perform the action. Later, B sends his treaty back to A, and A now has the original treaty again! This example demonstrates that, even when they are held by different users, treaties generated from the same original must be maintained in the same state.

Moreover, the situation can be seen to be even more profound when considering treaties which have been formed by the combination operations [15]. For instance, assume user A has received a treaty from B which allows him to do a write then a read, and has received another treaty from C that allows him to do a write then an execute, both referring to the same resource. If A were to do a join [15] of the two treaties and send

the result to D, and D were to request the kernel to perform a write action, whose treaty, B's or C's, is the correct one to be synchronized? Thus concurrency issues also affect the design and construction of treaty operations.

There is another area in treaty systems that should be aware of the duplication problem. The treaty combinator operations that are introduced in [14] and [15] provide a way to refine the behaviour specifications while they are holding by users. But it could cause problems if users would want to use a binary treaty operation to combine a treaty with itself, where the result could be the duplication of the original treaty. This is an issue especially for the operation of *concatenate*, which produces behaviours from the operands of the operation in sequences. If there is no particular concern to the issue, a user may gain extra actions by doing a combination of a treaty to itself, so that the result treaty would allow behaviours in this treaty happen twice in sequences.

It is probably necessary to have a look at what happens when binary treaty operations are performed to a treaty with itself. Having the given basic binary treaty operations *intersect* ($\alpha \sqcap \beta$), *join* ($\alpha \sqcup \beta$) and *difference* ($\alpha - \beta$), if we applies they with treaty α in both operands, the result would be:

$$\alpha \sqcap \alpha = \alpha$$

$$\alpha \sqcup \alpha = \alpha$$

$$\alpha - \alpha = \phi$$

where ϕ denote an empty treaty that does not provide any valid behaviours or actions to its holder, i.e. $[\![\phi]\!] = \{ \epsilon \}$. These could be proven using the formal definition of treaty operations and the concept of behaviour set:

$$[\![\alpha \sqcap \alpha]\!] = [\![\alpha]\!] \cap [\![\alpha]\!] = [\![\alpha]\!]$$

$$[\![\alpha \sqcup \alpha]\!] = [\![\alpha]\!] \cup [\![\alpha]\!] = [\![\alpha]\!]$$

$$[\![\alpha - \alpha]\!] = [\![\alpha]\!] \setminus \{ x \mid x = b.\Sigma^* \text{ where } b \in [\![\alpha]\!], b \neq \epsilon \} = \{ \epsilon \} = [\![\phi]\!]$$

And for the operation *concatenate* ($\alpha \cdot \beta$):

$$[\![\alpha \cdot \alpha]\!] = \{ x \mid x = b_1.b_2, \text{ where } b_1 \in [\![\alpha]\!] \text{ and } b_2 \in [\![\alpha]\!] \}$$

It is possible that $b_1 = b_2$, which causes the behaviour b_1 appears duplicated in the result. If b_1 contains one or more actions that have limited numbers of performing repetitions, this will leave the number of the action twice as much in the resulted treaty than that in the original treaty.

Hence it is reasonable to make the decision that one should never use binary treaty operations between a treaty and itself, as they only produce treaties that have no difference from the operands of the operations, or empty treaties that allow to do nothing, or produce treaties that break the assurance of behaviour control schemes.

Generally speaking, the most important new challenges of treaties are related to states, and their management in a scalable fashion in a distributed system. This is unsurprising, as behavioural states are exactly the novel feature that treaties provide.

To solve such a problem, it seems that central control will have to act as a key part. Critical information including the current state must be held somewhere that is controlled by the kernel, so that users cannot freely create independent copies without any restrictions. Although this may suffer some centralization problems, it still allows the property of treaty propagation, and kernels do not need to identify all users, hence it is not an unacceptable cost.

5. Models for Solving the Duplication problem

It has been proposed that treaties can be represented in various ways as long as they are still bounded to the conceptual definition of treaties [14]. Thus in practice there is more than one option to solve the duplication problem. However these solutions still have to follow the point that the crucial information of treaty states must be stored away from users. In the following part in this section there are three models of treaty representation to be introduced, aimed to solve the duplication problem.

5.1. Centrally Stored Treaties and References

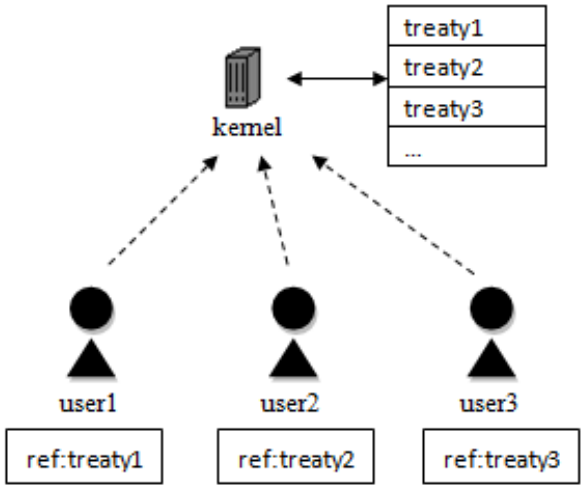

Fig. 2. Structure of a Centrally Stored Treaty System

A very simple way of solving the duplication problem is to take treaties away from users completely. In this case, users cannot access treaties directly any more, and thus are not able to make copies of treaties or send them to other users. Treaties will now be held in somewhere that can only be accessed by kernels. In return, users will only have references

to those treaties. In this approach, together with the behaviour descriptor and the current state and the reference to the target object, there will be a fourth part in treaties: a unique identifier for each treaty. The identifier will also be contained in treaty references held by users. Users can copy and send these references as many times as they like, as treaties are still kept unique on the kernel side. Fig. 2 shows an example of the structure of a treaty system that applies the representation of centrally stored treaties.

The process for preparing behavioural access control in such a model would be:

- A user process sends a request to the kernel, to create an object.
- The kernel creates the object, and creates a 'complete treaty' (unlimited accesses to all available action types for the object). This treaty will be stored in the treaty container that is managed by kernels, and the reference to this treaty, with the unique treaty identifier, will be given to the creator of the object.
- Then the creator can refine the behaviour descriptor by telling the kernel how he will use treaty operations.
- The kernel then constructs a new treaty and stores it in the container, and sends the new reference back.
- The creator can now send copies of this reference to other users, so that they share the access rights to the created object.

The process of performing an action on an object would be:

- A user needs to send a request together with the treaty reference.
- Upon receiving the request, the kernel then uses the identifier contained in the reference to locate the proper treaty, and checks whether the requested action is valid according to the behaviour descriptor in the treaty.
- If it is permitted, the action will be performed on the target object, otherwise the kernel will send a denying message implying that the action is not valid at this time.

This model is simple, but it has a weakness. In real cases it is quite possible that users do not know the current valid actions regarding the behaviour descriptors. And without access to the treaties, they will have to either $a)$ send access requests without any knowledge of treaties or, $b)$ send a query asking what types of actions are available at the moment. In both cases, the number of messages used for communication will increase.

However it needs to be mentioned that, in the case of option b, the returned 'available' types of actions might not be granted when the users try accessing objects. This is because there is another user who also has a reference to the same treaty, and he made an action using this treaty after the kernel returns the available action set and before the first user makes the accessing request. This is the non-deterministic problem that many concurrent systems may suffer, where processes share common resources and may disturb each other. A possible solution is to include semaphore scheme, but this is would be examined in future work.

5.2. User-held Treaties and Entries

In contrast to the previous implementation which locates treaties on the kernel side, this model lets users hold treaties, and kernels only keep a list containing a number of entries, each of which stores the information for a treaty. Treaties still have their unique identifiers, and they are contained in entries as well, so that kernels can link them to treaties.

Each entry consists of two parts: in addition to the treaty identifier, there is the current state of the related treaty. Although users can make copies and distribute treaties, these copies still have the same identifier and current state, which prevents the duplication problem happening. This approach also indicates that for a particular behaviour descriptor, all states are distinguishable. Each state is tagged to show the identity that makes the state different from other states in this behaviour descriptor. Fig. 3 shows an example of the structure of a treaty system that uses the entry list option.

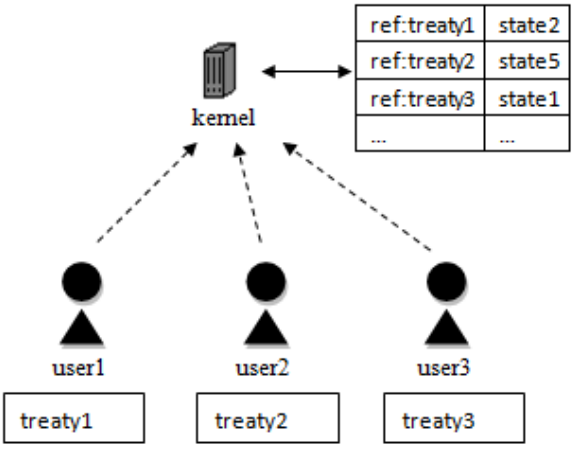

Fig. 3. Structure of a Treaty System with Entry List

To initiate the creation process:

- A user sends a request for creating a new object.
- The kernel creates the object and the complete treaty for the user, and an entry containing the identifier and the initial state for the treaty.
- The entry is added to the entry list, and the complete treaty is sent to the creator of the object.
- The creator can use treaty operations to modify the complete treaty by refinement.
- Before these refined treaties can be sent to other users, they must be first sent to the kernel for registration.
- The kernel assigns identifiers to these treaties, and records them by adding corresponding entries to the list.
- Now these treaties are ready for propagation.

If the user does not make the registration of their newly refined treaties to kernels, the kernels will not hold any information of these treaties. When users use these unregistered treaties to access objects, kernels simply deny these requests.

The process to access objects in this approach:

- A user needs to send the request with the correct treaty.
- When receiving the treaty in the request, the kernel first looks for its entry, and then checks whether the current state represented in the treaty is the same as represented in the entry.
- If the two states are the same, and the requested action is valid in this state, the kernel will update both states in the treaty and the entry to the new state according to the performed action, and return the treaty.
- If the two states are different, the kernel will synchronize them by updating the current state in the treaty with the state in the entry, and return this treaty to the user, notifying that this treaty has been updated.

Such an approach reduces the communication messages to some extent, as users can check the *possible* current state from treaties they are holding. Though there may still be cases where users get outdated information when the real state in the entry held by kernels has been changed by other users, it is suitable for applications which have few users sharing objects. This is also because when multiple users share the same object, the copy each of them holds is a treaty, which uses more space/memory, compared to references held by users in the previous approach.

5.3. Two-layered Referring FSM Representation

As indicated by the name, the representation chooses finite-state machines as the behaviour descriptor, hence each time a valid action is performed, there is an explicit state transition caused in relevant treaties. The key novelty of the Two-layered Referring FSM Representation is shown in its treaty storage location scheme. In this representation, treaties are neither stored centrally, nor completely distributed. Instead, there are two different categories of treaties in the system (See Fig. 4).

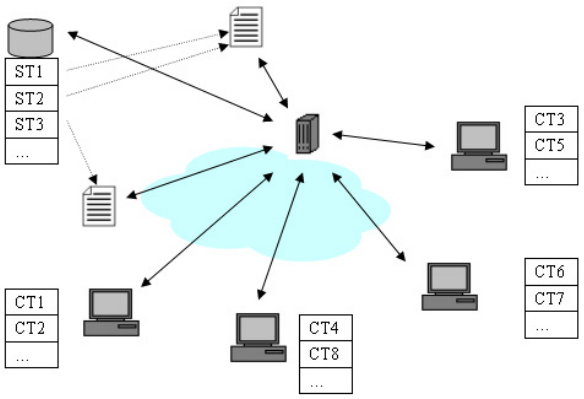

Fig. 4. Two-Layered FSM Representation

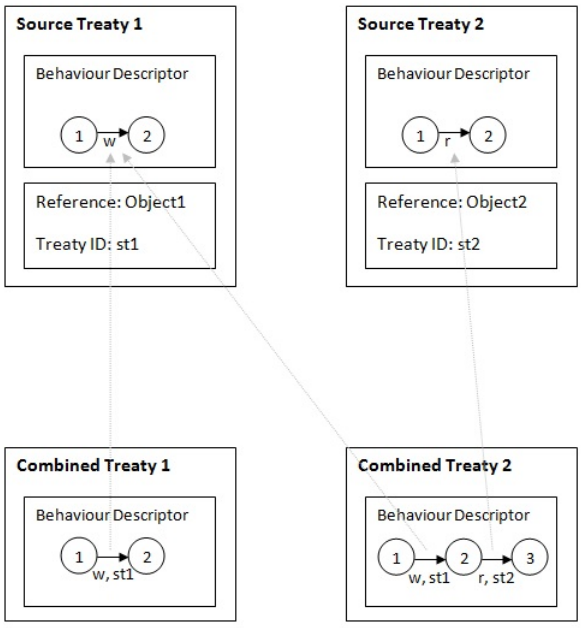

Fig. 5. Referencing Structure of Two-Layered Referring FSM Representation

The first kind is called ‘source treaties’. These treaties are stored on the kernel side, and each one can only refer to one object. Source treaties can be accessed by kernels and the creator of the referred object, but not any other users. Actions granted by the source treaties will be represented by transitions in the finite-state machine acting as the behaviour descriptor. The other kind of treaties is called ‘combined treaties’. These treaties are generated from source treaties, and they are held by users that are not the original creator of objects, but are permitted to have some accesses to those objects. Finite-state Machines are also contained in combined treaties, acting as behaviour descriptors, but combined treaties will not directly refer to any objects. Instead, each transition in the behaviour descriptors of combined treaties refers to one transition in a certain source treaty (See Fig. 5). When the holder of a combined treaty uses it to perform a permitted action on the targeted object, both the corresponding transition in the combined treaty and the referred transition in the source treaty will be processed.

To initialize, a process will apply to the kernel to create an object. The kernel creates the object, and returns a complete source treaty granting unlimited access for all action types that are available to this extension of object. After this, the creator of the object can use this complete source treaty to create treaties with more specified behaviour models, using treaty operations. At this stage all treaties created by the creator of the object are source treaties. When the behaviour model is obtained from a source treaty, the creator can save it in the storage of the kernel for distributed use. Each source treaty can be used as a model to build a combined treaty having exactly the same finite-state machine states

and transitions that represent the behaviour. Each transition in the generated combined treaty refers to the original transition in the source treaty.

After this, the creator can send out copies of combined treaties, and these copies are built from the source treaties this creator is holding. When other users have received many combined treaties, they can use treaties operations to refine the behaviour model to generate new combined treaties and propagate them out as well. This also indicates that there may be transitions in a same combined treaty that refer to source treaties of different object, which means that the referencing style of the Two-layered Referring FSM Representation is mixed multiple referencing [14].

The reason for this design is that, however users make copies and propagate to duplicate combined treaties, their transitions will still refer to the transition in the source treaties (which are not duplicated). After a user performed an action, other users may not be able to do the same action because the transition in the source treaty that grants this access has already occurred. In this way, the duplication problem has been solved in this representation.

6. Evaluating Performances

We have proposed several models that are designed for solving the duplication problem, and they are going to be examined and compared for their performance with respect to time and memory consumption. The treaties and related components were implemented in Java, and they were installed in a discrete event simulation tool (SimJava [6]) which is also used for network and distributed environment simulation. A use-case of voting system is chosen here. Three options have been given for voting, and the behaviour model for each voting treaty is shown in Fig. 6.

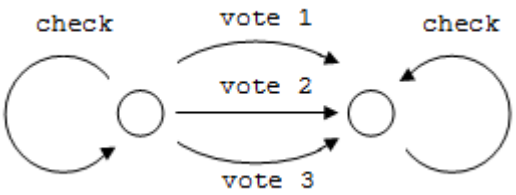

Fig. 6. Behaviour Model for a Vote Treaty

There are kernels and users who are communicating with events scheduled, including requests from users and acknowledgments from kernels. There is an object counting the voting result, and it can only be accessed by kernels. We used methods `gc()`, `totalMemory()` and `freeMemory()` in the `Runtime` class for garbage collection and memory calculation. When processing, each user starts its job by sending a request for creating a treaty to the kernel. Upon reception of the treaty (or treaty reference), the user chooses one of the three options to vote, and the kernel then changes the result in the counting object, until all requests have been made and processed.

Fig. 7 shows the time consumption when using one kernel to process different numbers of voters in our three models. The horizontal axis is the number of voters and the vertical axis is the total time consumed in milliseconds. From this it can be observed that the time spent by the Central Store approach and the Entry List approach is almost the same, while the Two-Layered Treaty approach takes about 40% more. Fig. 8 shows the time consumption when using multiple kernels to process 100 voters in the three models. The result for multiple kernels does not show a great improvement from single kernel processing, as the total amount of work by all kernels is nearly unchanged.

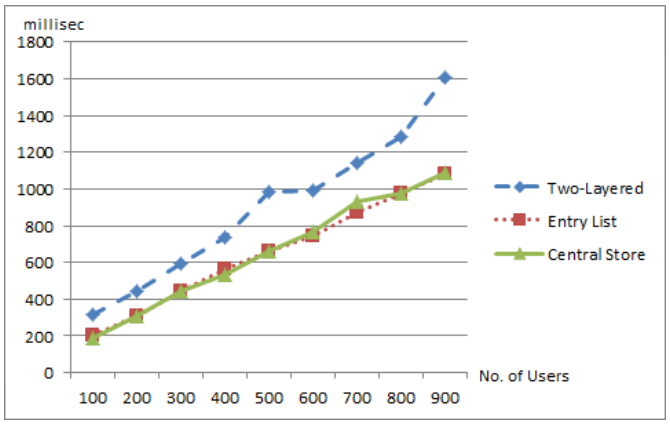

Fig. 7. Time Consumption while Number of Users Changes

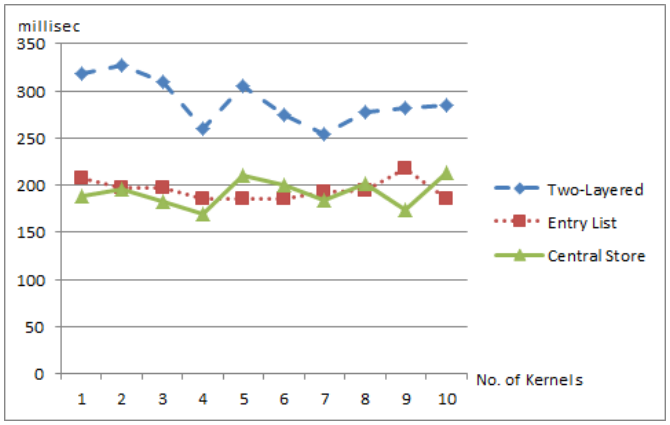

Fig. 8. Time Consumption while Number of Kernels Changes

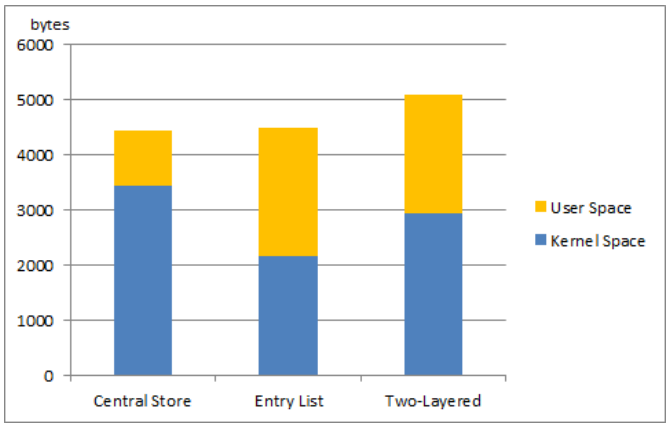

Fig. 9. Space Consumption Comparison in Small Scaled Case

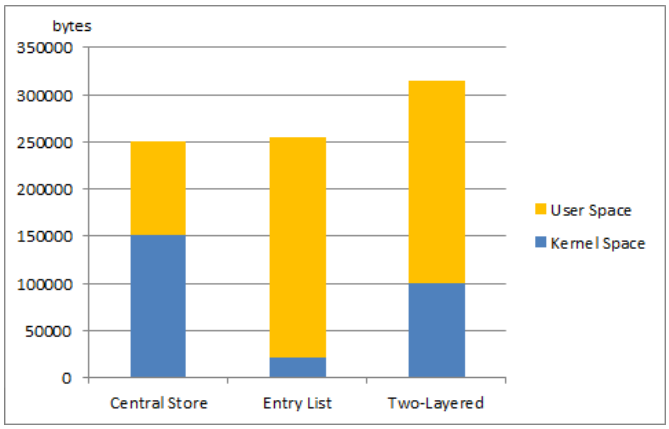

Fig. 10. Space Consumption Comparison in Larger Scaled Case

Fig. 9 shows the memory consumption when having one kernel and one user, with one treaty. The vertical axis is the memory consumed in bytes. In the Central Store approach most of the memory consumption lies on the kernel side, while in the Entry List approach there is a balanced state. The Two-Layered treaty approach costs more memory space than the other two. Fig. 10 shows the memory consumption when there are 10 kernels and 100 users with 100 treaties. We can see that in the Central Store approach, the kernel load is very heavy, as it still takes more memory than the sum of user space. In the Entry List approach, most of the memory consumption goes with user processes. Again the Two-Layered treaty approach costs more memory space than the other two, but it should be noted that this approach has the extra functionality of modelling sequences among multiple objects.

7. Evaluating Scalability

The scalability of treaty systems is a very important measurement in this project, as treaties are proposed to work in distributed environments where the number of users is dynamic and varies widely. However, there is a problem before any evaluation can be carried: the concept of scalability has not yet been formally defined. Scalability in parallel computing was originally used to describe the enhancement of processing speed when multiple processing units are involved in a computation. But as the areas of parallel and distributed computing develop and applications and services in these fields keep varying, there is no single, commonly agreed way of measuring the 'scalability' for different systems.

Thus it is best to define the meaning of scalability in treaty systems before carrying out any evaluations. The chosen measurement of 'scalability' shall be suitable for illustrating the performance of systems when the number of kernels and users changes in distributed environments. We define treaty system to be 'scalable' if the consumption of key measurements of the system increases linearly (or approximately linearly) while the population in the system increases linearly, which means the ratio between increased amount of consumption for resource and the increased amount of entities in the system is approximately constant.

There are two major measurements that are selected for evaluating the performance of treaty systems, they are *the additional resource on kernels* and *the processing time for requests*. The additional resource on kernels is mainly used for storing state information of treaties, as it has been explained that there must be some information stored by kernels to solve the duplication problem. Ideally, the resource should have a linear increase with the number of behaviour models, and we can certainly call treaty systems in this situation 'scalable'. However, in practice it is always hard (or impossible) to reach the ideal case. We will need to trace the trend of increases of resources in kernels as the number of users increases (assuming each of them sets up a separate treaty).

The processing time for requests is another measurement that can reflect the performance of treaty systems in distributed environments. In this part of evaluation we would like to examine the average processing time for each request, which is calculated by having the overall processing time divided by the number of requests. We will trace changes in this average value when users and requests keep growing to evaluate the scalability on processing time. This differs from the evaluation in the previous section which compares

the processing time among capabilities, vistas and treaties. In the ideal case, the increase in average processing time for a single request should also be linear.

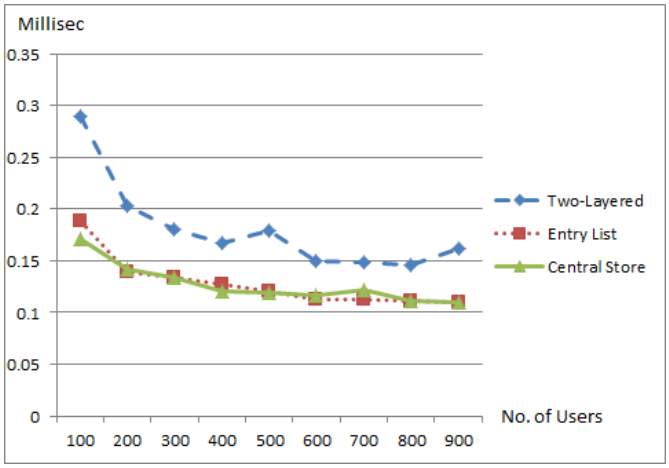

Fig. 11. Average Time Consumption per Message while Users Increase

Fig. 11 shows the average time per message when users in the system increase. We still do the comparison among the three models that aim to solve the duplication problem. The messages that are counted in this experiment includes the request for obtaining a treaty, the request for accessing the object using the treaty, and the acknowledgment message for these two kinds of request. We can see from the figure that as the experiments go, there are fewer factors that disturb the evaluation result, so the average time consumption for processing each message becomes steady in later stages. This shows that when the number of users increases, the processing time for requests is not much affected, which implies good scalability in time consumption.

Fig. 12 shows the kernel space occupation when the number of users in the system increases. From the result we can see that all the three models have a linear increase in the space occupation on the kernel side. This also shows that treaty systems have a good scalability in additional resource consumption on kernels. But note that in the three approaches, the entry list approach has the lowest resource occupation among the three, which seems to be the better choice if it is suitable to applications.

8. Conclusion

In this paper we have reviewed the concept of treaties as a behaviour control approach, which have a number of operations [15] that can be used to refine these behaviour models. The formal definitions of these operations are given. We have also discussed the duplication problem which is an important issue in treaty systems. Then three models of treaty systems were introduced. Evaluation results shows the time and memory consumption of the three models in a use-case, illustrating that the central-store approach consumes

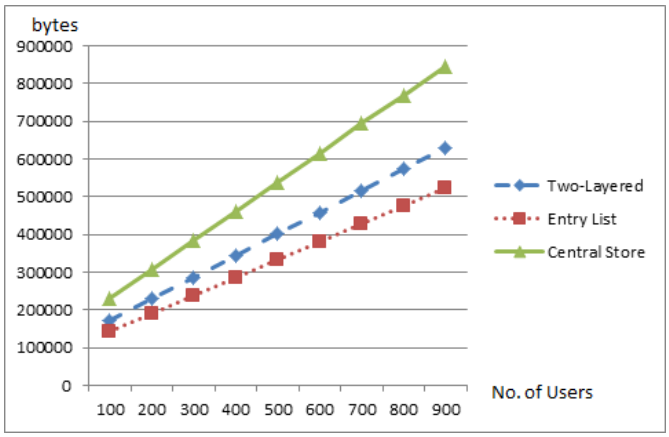

Fig. 12. Kernel Memory Occupation while Users Increase

more kernel resources while the entry-list approach consumes more on user side, and the two-layered treaty approach has a more balanced situation but consumes more resources totally. The result of evaluating the scalability also shown that treaty systems are 'scalable' in given measurements, as the time for processing each access request does not increase, and the memory consumption on kernel side increases linearly.

Future work will involve investigating other representations of behaviour descriptors, such as regular expressions, and comparisons between performances of the two behaviour descriptors will be examined. We will also work on summarizing laws for treaty operations.

References

1. Dennis, J.B., Horn, E.C.V.: Programming semantics for multiprogrammed computations. Communications of the ACM (3), 143–155 (3 1966)
2. Gay, S., Vasconcelos, V., Ravara, A.: Session types for inter-process communication. Tech. rep., Department of Computing, University of Glasgow (3 2003)
3. Gelernter, D.: Generative communication in linda. ACM Transactions on Programming Languages and Systems (1), 80–112 (1 1985)
4. Gong, L.: A secure identity-based capability system. In: Proceedings of 1989 IEEE Symposium on Security and Privacy. pp. 56–63 (1989)
5. Gorla, D., Pugliese, R.: Dynamic management of capabilities in a network aware coordination language. Journal of Logic and Algebraic Programming 78(8), 665 – 689 (2009)
6. Howell, F., McNab, R.: Simjava: A discrete event simulation library for java. In: Simulation Series. vol. 30, pp. 51–56 (1998)
7. Karp, A.H., Rozas, G.J., Banerj, A., Guptai, R.: Using split capabilities for access control. IEEE Software (1), 42–49 (1 2003)
8. Lampson, B.W.: A note on the confinement problem. Communications of the ACM (10), 613–615 (10 1973)
9. Miller, M.S.: Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. Ph.D. thesis, Johns Hopkins University, Baltimore, Maryland, USA (May 2006)

10. Shapiro, J.S., Smith, J.M., Farber, D.J.: Eros: a fast capability system. In: Symposium on Operating Systems Principles. pp. 170–185 (1999)
11. Udzir, N.I.: Capability-Based Coordination For Open Distributed Systems. Ph.D. thesis, University of York - Department of Computer Science (2007)
12. Wood, A.: Coordination with attributes. In: LNCS 1594 Coordination Languages and Models, COORDINATION '99. pp. 21–36 (1999)
13. Wood, A., Zhao, Y.: Vistas: towards behavioural cloud control. In: Euro-Par 2010 Parallel Processing Workshops. pp. 689–696 (2011)
14. Zhao, Y., Wood, A.: Treaties: Behaviour-controlling capabilities. In: 2nd Annual International Conference on Advances in Distributed and Parallel Computing (ADPC 2011). pp. 19–24. GSTF (2011)
15. Zhao, Y., Wood, A.: Behavioural Sets and Operations in Treaty Systems. In: Proc. International Conference on Control Engineering and Communication Technology (2012)

Yining Zhao received his Bachelor and Master degree of Computer Science in the University of Manchester in 2006 and 2007 respectively. In 2013 he obtained his Ph.D. degree in the University of York. His researching activities and interests are in the area of distributed-computing and related topics, and in particular the systems and security issues in these environments.

Alan Wood graduated with a B.Sc. in Electronics from the University of Kent UK, and then pursued research in Image Processing and Parallel Computing at University College London, obtaining a Ph.D. in 1983. He then worked as a Lecturer in the Department of Physics at UCL as part of the team which developed the CLIP series of parallel array processors before taking up a Lectureship in the Department of Computer Science at the University of York in 1989. At York Dr Wood developed a research group investigating Coordination Languages focussing on the Linda tuple-space paradigm. This has led to interests in languages for concurrency, as well as the issues associated with open access in distributed systems.

Received: February 4, 2013; Accepted: October 24, 2013.