Highlights

Information Management for Trust Computation on Resource-constrained IoT Devices

Matthew Bradbury, Arshad Jhumka, Tim Watson

- Identify the problem of managing information used to evaluate trust on highly memory constrained IoT devices
- Prove that this problem is NP-complete
- Propose a trust model-agnostic utility function to quantify the usefulness of information to trust evaluation
- Propose novel eviction strategies targeting trust information storage
- Simulate a large number of eviction heuristics to explore their performance
- Solve the optimisation problem of how to allocate memory to different memory regions at compile time

Information Management for Trust Computation on Resource-constrained IoT Devices

Matthew Bradbury[a,*], Arshad Jhumka[b], Tim Watson[c]

[a]School of Computing and Communications, Lancaster University, LA1 4YW, Lancaster, UK
[b]Department of Computer Science, University of Warwick, CV4 7AL, Coventry, UK
[c]WMG, University of Warwick, CV4 7AL, Coventry, UK

Abstract

Resource-constrained Internet of Things (IoT) devices are executing increasingly sophisticated applications that may require computational or memory intensive tasks to be executed. Due to their resource constraints, IoT devices may be unable to compute these tasks and will offload them to more powerful resource-rich edge nodes. However, as edge nodes may not necessarily behave as expected, an IoT device needs to be able to select which edge node should execute its tasks. This selection problem can be addressed by using a measure of *behavioural trust* of the edge nodes delivering a correct response, based on *historical* information about past interactions with edge nodes that are stored in memory. However, due to their constrained memory capacity, IoT devices will only be able to store a limited amount of trust information, thereby requiring an *eviction* strategy when its memory is full of which there has been limited investigation in the literature. To address this, we develop the concept of the memory profile of an agent and that profile's utility. We formalise the profile eviction problem in a unified profile memory model and show it is NP-complete. To circumvent the inherent complexity, we study the performance of eviction algorithms in a partitioned profile memory model using our utility metric. Our results show that localised eviction strategies which only consider one specific type of information do not perform well. Thus we propose a novel eviction strategy that globally considers all types of trust information stored and we show that it outperforms local eviction strategies for the majority of memory sizes and agent behaviours. In this paper, we develop a concept of information utility to a trust model and formalise the problem of information eviction, which we prove to be NP-complete. We then investigate the usefulness of different eviction strategies to maximise the utility of information stored to enable trust-based task offloading.

Keywords: Trust, Information Management, IoT, Resource-constraints, edge nodes, offloading

1. Introduction

Resource-constrained Internet of Things (IoT) devices (henceforth called IoT devices) have previously been used to perform sensing and actuation after receiving commands from a server. However, there is increasing interest in having those resource-constrained devices autonomously performing actions. With the limited resources these devices have, they will not be able to perform expensive computation on their own and will need to interact with other devices to complete some computation, such as by offloading tasks to resource-rich edge nodes (henceforth called edge nodes) [1].

For redundancy multiple edge nodes should be provisioned in the network, which means that there is a need for IoT devices to select which edge node should be interacted with. However, there is a risk that interactions may not be correct or successful, e.g., due to network attacks, edge nodes being overloaded with tasks, or edge nodes intentionally misbehaving. One approach to mitigate this threat is to use a measure of *behavioural trust* to select which edge node to offload a task to. This measure of behavioural trust can be formed by recording opinions of past interactions with other nodes such that *trust models* (such as the Beta Reputation System [2]) can use these records to quantitatively assess whether nodes will provide services correctly in the future.

An issue with IoT devices is that they have limited memory, so there exists a trade-off when storing historical interaction information. Will IoT devices obtain a higher utility by storing lots of information about a few select nodes or a little information about many nodes? For small systems and trust models with small memory footprints, all information can be stored, but, in larger systems it will not be possible to store information about all agents in the system. This problem is further complicated by applications for IoT devices avoiding dynamic memory allocation to prevent memory fragmentation [3]. Instead, memory is divided up into pools of one type and given a static size decided at compile time. These issues mean that the management of interaction information used for future interaction success prediction is important. Specifically, there are two key research questions that we investigate in this work:

1. When a new item needs to be stored and memory is full, which existing item should be evicted?
2. How should a small amount of memory be divided up to store different types of information used to evaluate trust?

*Corresponding author
Email addresses: M.S.Bradbury@lancaster.ac.uk (Matthew Bradbury), H.A.Jhumka@warwick.ac.uk (Arshad Jhumka), tw@warwick.ac.uk (Tim Watson)

A wrong eviction may reduce the ability of a device to perform a successful interaction, because unlike evictions in database, disk or network buffers, records of interactions used to build a trust model *cannot* be recreated. This means that a wrong eviction can lead to necessary information not being present in memory when required. In addition, because dynamic memory allocation is avoided, eviction of arbitrary data from memory for new information is not possible, instead a specific item with the matching type needs to be chosen for eviction.

Therefore, in this paper we investigate the two problems identified by making these five contributions:

1. we introduce the notion of an agent's memory profile and its utility, we define and formalise the memory-constrained profile selection (MyCoPS) problem, and show that it is NP-complete in a unified profile memory model,
2. to circumvent the inherent exponential complexity, we implement a partitioned profile memory model to which several existing eviction algorithms (such as Least Recently Used (LRU) and others) can be applied,
3. we propose new eviction strategies that consider information in single memory regions (i.e., local algorithms) and also information across multiple memory regions (i.e., global algorithms),
4. we perform a simulation of agents with limited resources interacting with other agent's services and investigate the performance of multiple commonly used eviction strategies plus our proposed eviction strategies on maintaining information used to evaluate if future interactions may be successful, and
5. we use IBM's ILOG CPLEX to solve the optimisation problem of how to size statically allocated memory regions containing information used for trust evaluation.

The rest of this paper is structured as follows: related work on trust models and cache replacement is presented in Section 2. Section 3 describes the system, fault and trust models. How agent's profiles are managed is described in Section 4, the problem's complexity is proved in Section 5, and how memory can be partitioned to better manage profiles on agents is described in Section 6. Example memory contents, the eviction strategies explored and our proposed metric to evaluate the *utility* of memory contents are described in Section 7. Section 8 describes the experimental setup used to generate the results in Section 9 which evaluate the performance of the eviction strategies via the utility metric. Optimal ways to size memory regions are presented in Section 10. A discussion of these results is presented in Section 11 and we conclude in Section 12.

2. Related Work

Trust has a variety of different meanings and those which are relevant depend on the context in which trust is being used. One example, is *identity trust*, where an agent is capable of verifying the authenticity of a message's sender. Identity trust can be provided via mechanisms such as digital certificates [4]. However, in this work we are concerned with *behavioural trust* where past interactions with a trustee are used to build a subjective belief of the likely outcomes of future interactions from the perspective of a trustor [2, 5].

2.1. Trust Representation

The Beta Reputation System (BRS) [2] models how a trustee is expected to behave based on past interactions that are classified as either good or bad using the Beta distribution. The Beta distribution is composed of the number of good interactions α and the number of bad interactions β. The expected value of the Beta distribution ($^{\alpha}/_{\alpha+\beta}$) predicts the likelihood of a good future interaction. BRS can be extended to multiple dimensions using the Dirichlet distribution [6]. Other distributions such as Gaussian are also used [6].

BRS either supports or has been extended to have additional features such as forgetting [7], discounting unreliable feedback (TRAVOS [8]), and combining feedback. However, a limitation is the assumption that agents will tend to perform the same over time. Using a finite state Hidden Markov Model (HMM) allows more dynamic system behaviour to be captured [9].

Both BRS and HMM-based trust models are relatively simple and their advantage is in their small size. There are a number of other trust models that use more information sources to improve their decision making. Bayesian networks [6, 10] allow handling and combining multiple aspects of trust via a probability relation. A downside is that they can be large if there are many elements being considered. They can also be expensive to compute, especially on hardware without support for floating point operations. There are many other ways of structuring trust models, including decision trees [11], support vector machines [12], Q-learning models [13] and managing the information in structures such as a blockchain [14].

In order to form an opinion, an interaction does not need to be performed with a trustee. Instead interactions between other agents can be overheard and used to form an opinion [15], or alternatively, agents can form initial opinions based on stereotypes [16] in order to bootstrap trust evaluation when there has been little initial interaction. Another option is to use a proactive assessment of trust [17] which requires little memory, but comes with a trade-off in terms of accuracy and higher computational costs. Other approaches may not even involve assessing a measure of trust. For example, a reward-based scheme [18] instead incentives good behaviour by rewarding agents that perform interactions correctly.

2.2. Information Management

Information management has been extensively studied in terms of cache replacement policies on the web [19], wireless dissemination [20], databases [21], and page replacement algorithms [22]. This was an issue explored by Chen et al. [23] in Section 4.4 for trust models, where the authors assume that agents with the highest trust values (top 50%) and recent interactions are the most important to keep. If the memory is full, the record with a trust value lower than the median and was the earliest interaction, is evicted. This work considers a data structure that only manages trust information calculated from interactions.

In order to avoid querying all agents in a system for their trust data, PeerTrust [24] maintained a cache of the "most recent trust values of other peers [the peer] has interacted with in the past". If a peer's trust value is not present in the cache "a default value" is used instead. This means that a least recently used (LRU) cache replacement policy is used to evict items when the cache is full.

Alternate mechanisms involve using a distributed data store, such as a blockchain [25] to record transactions. This record could be used to decide where a task should be offloaded to. However, there exists a trade-off in terms of how much of the blockchain should be stored locally in limited memory and how much energy should be spent (in terms of communication) keeping up-to-date with transactions.

There has been little evaluation in the literature of managing the contents of memory used to evaluate trust. This is problematic because in the event of incorrect information used to evaluate trust being held in memory means there is not just the potential for reduced performance in terms of receiving a slower or less accurate result from an interaction with another agent, but for autonomous systems to make worse decisions than if good trust information on historical actions was retained.

3. Models

In this section, we present the models we assume in this system. This includes (i) the assumptions we make about the system, (ii) how agents interact, (iii) how interactions between agents can fail, and (iv) how agents will assess which other agents to interact with based on historical observations.

3.1. System Model

We assume a network of IoT agents represented as the graph $G = (V, E)$, where each agent $n \in V$ has a set of computation resources to execute tasks and the ability to perform interactions with other agents. An edge $(n, m) \in E$ exists if agent n can directly communicate with agent m. We assume the network is connected, i.e., any pair of agents can communicate over a single-hop or multi-hop route. Resource-constrained agents are not assumed to be equipped with any stable storage due to the energy cost [26]. We assume the network is synchronous, i.e., there are known bounds on network latency and processor speed.

Resource-constrained devices may wish to offload to resource-rich edge nodes those expensive tasks they lack the ability to execute. We model this by having each agent n in the network have a set S_n of services that it provides. Resource-constrained devices may need to perform multiple types of interactions, so resource-rich nodes need to provide multiple services that can process these interactions.

There is a large variety of relevant IoT hardware with limited resources. Representative hardware includes the Zolertia RE-Mote [27] with a 32 MHz CPU, 32 KiB of RAM, 512 KiB of programmable flash, a 800 mAh battery, and optionally an SD card. Or the nRF52840 [28] SoC which has a 64 MHz CPU, 256 KiB of RAM and 1 MiB of programmable flash. These devices are equipped with hardware acceleration for public key cryptography operations, making the of public key cryptography operating such as shared secret derivation from public keys suitable.

3.2. Agent Interactions

Every agent has three types of actions that it can execute: (i) *Dissemination* ($\mathfrak{D}$), which allows an agent to send data to other agent(s) in the network, (ii) *Request* ($\mathfrak{R}$), which allows an agent to ask another agent for information, and (iii) *Execution* ($\mathfrak{E}$), which allows an agent n to require another agent m to execute one of its services and then deliver the output to n. Information gathered during an interaction with another agent is stored in a relevant region of memory in order to evaluate trust.

Definition 3.1 (Interaction)**.** When an agent n performs an action $a \in \{\mathfrak{D}, \mathfrak{R}, \mathfrak{E}\}$ intended towards an agent m, where $n \neq m$, we say that n *interacted* with m.

3.3. Fault Model

We assume that an agent can fail *arbitrarily* when performing the execution action. This failure may be *spatially* and/or *temporally* during execution [29]. For example, an agent may execute a service incorrectly by returning the wrong result or returning a correct result at the wrong time. We assume that an agent will perform the dissemination and request actions correctly.

3.4. Trust Model

The agents in the system each maintain a trust model that contains information such as historical interactions, peer-provided reputation and stereotypes to assess trust in other agents. Each agent maintains its own trust model and uses it to decide which agent an interaction should be performed with. We expect that agents will use a combination of the following to evaluate trust based on historical interactions: (i) whether a service request is accepted, (ii) whether a result is returned within some deadline, and (iii) whether the result is correct [30]. It may not always be possible to evaluate if a result is correct (due to resource-constraints) so we assume there to be a lightweight check (i.e., a simple predicate) to validate the answer.

We assume that each agent collects the same kind of information about other agents. However, agents may not know the same information. For example, an agent may not have any direct trust information on another agent due to a lack of interaction between them. Agents may not have public keys for other agents, and so may be unable to interact with them unless those public keys are first obtained.

4. Agent Profiles Management for Trust Model

As agents interact they will store information about these interactions. For example, a trust model running on such resource-constrained devices will use the allocated memory to store trust-relevant information that are subsequently to be used for trust computation, such as: agents in the network, services offered by other agents, digital certificates, and other information. These pieces of information are represented as individual *items*.

Definition 4.1 (Item)**.** An item is a piece of information of a specific type. Each item comes from a specific set of values. The item I has size $\textsc{SizeI}(I)$. All items from the same set of values have the same size.

Example items may include: a service that agent n provides, trust information that agent n holds about another agent m, stereotypes about agent n's expected behaviour, a digital certificate for an agent n, or other information relevant to secure communication or trust evaluation.

The information an item contains may refer to one or more agents. There exists a function called $\textsc{Subjects}()$ which takes a given item and returns the set of agents to which that item refers to. For example, an item I that contains reputation values that agent j holds for multiple other agents, $\textsc{Subjects}(I)$ will return those multiple other agents.

When an agent n interacts with another agent m, one of those interactions can involve n sending a dissemination message to m that contains multiple information items.

Definition 4.2 (Dissemination Message)**.** A dissemination message $D_{n\to m}$ sent from agent n to agent m is a message with k items. The i^{th} item of the dissemination message is denoted $D_{n\to m}[i]$. $D_{n\to m}[i] = \perp$ denotes that the i^{th} item is null and contains no information.

When an agent m receives a message $D_{n\to m}$ from another agent n, m needs to extract the relevant information from $D_{n\to m}$ and update the information it has about other agents referenced to in $D_{n\to m}$. As such, we define the concept of a profile of an agent o stored in agent m's memory.

Definition 4.3 (Profile)**.** A profile is an l-tuple, which holds l items on a single agent. The set of all possible agent profiles is denoted by $\mathbb{P}$.

The size of a profile p_n of agent n is denoted by $\textsc{SizeP}(p_n)$. This size can be found by summing the sizes of items in the profile $\textsc{SizeP}(p_n) = \sum_{I\in p_n} \textsc{SizeI}(I)$.

Definition 4.4 (Profile Memory)**.** The profile memory of an agent n, denoted by M^n, is a set of profiles that agent n has about 0 or more agents, i.e., $M^n = \{\, m \mapsto p_m \mid \forall m \in V, m \neq n \,\}$.

To highlight that each agent has a set in M^n we use the notation $m \mapsto p_m$ to represent that p_m is the profile for agent m. We will use this notion for other definitions to specify that an item in a set is an item for a particular agent or item type.

So in summary, an agent n maintains a profile memory M^n which contains a set of profiles. Each profile in the profile memory holds a tuple of items, where each item in a single profile has a single subject that is the same for all items in this profile. Each item has a specific type, which specifies what kind of information it contains. An example profile memory is shown in Figure 1 where agent A10 stores information on 5 agents where each of the profiles are made up of three items. Some of these items are empty. Dissemination messages are used to transmit items from one agent to another agent.

Example Profile Memory stored on Agent A10

Profile A1	I1	I2: C1, C2	I3: A3, A4, A5
Profile A3	I1	I2: C2	
Profile A4		I2: C1, C2	
Profile A5	I1		I3: A3, A4
Profile A9	I1		I3: A5

Figure 1: Example profile memory labelled with the item type of the information held in a partitioned memory model. Item 2 can hold information on each of the services provided — C1 and C2. Item 3 indicates which agents provided this information on the agent who is the subject of the profile.

On reception of a dissemination message, a function $\textsc{Extract}$ is used to extract profiles from the items contained in the message. This means that given a dissemination message $D_{n\to m}$ which holds multiple items on potentially different agents, $\textsc{Extract}$ uses $D_{n\to m}$ to obtain a set of profiles. These agent profiles are then used to update the profile memory on agent m. The updating of agent m's profile memory with profile p_o of agent o is denoted by $M^m := M^m \oplus \{\, o \mapsto p_o \,\}$. The agent m's profile memory can be updated with multiple profiles simultaneously.

Similar to a dissemination message, a profile may not contain every possible item. Therefore, there exists an important difference between a partial profile and a complete profile. We assume that some items within a profile are crucial, i.e., the lack of these items will mean interaction between agents m and n is not possible. Therefore, the absence of these crucial items in a partial profile will render that profile useless.

When further messages are received by m, then the profile may be updated, e.g., through aggregation. However, due to the memory constraints of an agent, it may be impossible for an agent m to keep profiles of all other agents in the network. Therefore, when an agent m receives a message from a new agent n when its profile memory is already full, m needs to decide which profile to remove from memory to be replaced by that new profile (or to discard the new profile).

We thus define the notion of a constrained profile memory[1].

Definition 4.5 (Constrained Profile Memory)**.** Given a value Δ, the profile memory M^n of an agent n is said to be constrained if the size of M^n can never exceed Δ. We say that profile memory M^n is full if there is no profile that can be combined with the profile memory such that it is within the size constraints, i.e., $\forall p_m \in \mathbb{P}, p_m \notin M^n \implies \textsc{SizeP}(M^n \oplus \{m \mapsto p_m\}) > \Delta$.

We now provide the definitions of insertion, removal and eviction. For these definitions, the state of the dedicated memory area at agent n before the execution of an action is denoted by M^n and the state after the execution of the action by $\bigcirc M^n$.

Definition 4.6 (Insertion)**.** Given an agent n with memory M^n and a new profile p_m of agent m, we say that profile p_m is inserted

[1] Henceforth, we mean constrained profile memory whenever we say profile memory.

into a memory region M^n if $((\{m \mapsto p_m\} \notin M^n) \wedge (\bigcirc M^n = M^n \cup \{m \mapsto p_m\}))$.

Definition 4.7 (Removal)**.** Given an agent n with memory M^n, we say that profile p_m of agent m is removed from M^n if $((\{m \mapsto p_m\} \in M^n) \wedge (\bigcirc M^n = M^n \setminus \{m \mapsto p_m\}))$.

Definition 4.8 (Eviction)**.** Given an agent n with memory M^n, we say that profile p_m of agent m is evicted for profile p_o of agent o in a full profile memory M^n if p_m is removed from M^n and profile p_o is inserted into M^n, i.e., $((\{m \mapsto p_m\} \in M^n) \wedge (\{o \mapsto p_o\} \notin M^n) \wedge (\bigcirc M^n = (M^n \setminus \{m \mapsto p_m\}) \cup \{o \mapsto p_o\}))$. When such a profile p_m for agent m exists, we will only say that profile p_o for agent o is evicted.

At any point in time, an agent n may be receiving messages from multiple agents. We assume that there exists a finite-sized buffer at n that stores all received messages before they are processed. The order in which the messages in the buffer are processed may vary according to the application. For example, some applications may process the messages in first-in first-out (FIFO) order, others may be based on priority, or other suitable schemes. Messages may be processed in groups to enable potentially better decision making.

When these messages are processed to update individual agent profiles, if profile memory area is not full then the addition of a new profile is trivial as space exists for it. A challenge exists when the profile memory is full as one or more evictions will be required to happen. As such, an objective can be to keep only those profiles with the highest *utility* values which we quantify in Section 7. But first, we study the complexity of this problem to understand potential ways to solve this problem.

5. Profile Management Complexity

We now provide a formalisation of the problem we pose in this paper, which we call as the Memory-constrained Profile Selection Problem (MyCoPS). Firstly, we need to gauge the utility that a profile provides. We assume that the more items contained in agent n's profile of another agent m, the greater the utility of that profile.

Definition 5.1 (Memory-constrained Profile Selection (MyCoPS))**.** Given an agent n with:

- a full profile memory M^n of size Δ,
- a set of new messages P,
- a minimum threshold for profile memory utility $\mathcal{V}$, and
- a function $\textsc{SizeP}(p)$ that returns the size of a profile p and a function $\textsc{Utility}(p)$ that returns the utility of a profile p,

select a set $P' \subseteq (M^n \oplus \bigcup_{p \in P} \textsc{Extract}(p))$ such that

$$\sum_{p \in P'} \textsc{Utility}(p) \geq \mathcal{V} \wedge \sum_{p \in P'} \textsc{SizeP}(p) \leq \Delta \, .$$

In this section, we show that the MyCoPS problem is NP-complete by showing a reduction of 0-1 Knapsack [31, p. 134–139] to MyCoPS. The intuition is to only keep profiles that increase the utility of the memory content.

Definition 5.2 (0-1 Knapsack)**.** Given a set I of n items labelled $I_1, \ldots, I_n$, with each item I_j having a weight w_j and value z_j, a maximum weight value W and minimum value Z, is there a set $I' \subseteq I$ such that

$$\sum_{I_j \in I'} z_j \geq Z \wedge \sum_{I_j \in I'} w_j \leq W \, .$$

Lemma 1 (MyCoPS and class of NP)**.** *MyCoPS is in NP.*

Proof. To prove this, we need to verify the correctness of a possible solution set P' in polynomial-time. So, given an instance of MyCoPS and a solution set P', we verify the correctness of P' as follows:

1. Computing the total utility of profiles in P' is done in linear time.
2. Computing the total size of profiles in P' is done in linear time.
3. The two conditions can be trivially verified.

The complexity of this verification procedure is $O(|P'|)$. □

Lemma 2 (MyCoPS and NP hardness)**.** *MyCoPS is NP-hard.*

Proof. To prove this, we first map and then reduce 01KP decision problem to the MyCoPS problem.

- $(M^n \oplus \bigcup_{p \in P} \textsc{Extract}(p)) \mapsto I$
- $P' \mapsto I'$
- $\Delta \mapsto W$
- $\mathcal{V} \mapsto Z$
- $p_i \mapsto i$
- $\textsc{SizeP}(p_i) \mapsto w_i$
- $\textsc{Utility}(p_i) \mapsto z_i$

Reduction: We now have to show that a solution to 01KP exists if and only if a solution of MyCoPS exists.

⇒ Let I' be a solution to the 01KP problem. Let $\mathcal{J}'$ be a solution to the instance of MyCoPS as defined under the mapping such that $I' = \mathcal{J}'$. We show that $\mathcal{J}'$ is a valid solution for MyCoPS.

In set $\mathcal{J}'$, we include all profiles p_i such that $\textsc{SizeP}(p_i) \mapsto w_i$, with $i \in I'$. Thus,

$$\sum_{p_i \in P'} \textsc{SizeP}(p_i) \leq W \implies \sum_{p_i \in P'} \textsc{SizeP}(p_i) \leq \Delta \, ,$$

under the identified mapping. Also, since $p_i \in P'$, it means that

$$\sum_{p_i \in P'} \textsc{Utility}(p_i) \geq Z \implies \sum_{p_i \in P'} \textsc{Utility}(p_i) \geq \mathcal{V} \, .$$

Hence, $\mathcal{J}'$ is valid.

⇐ Let $\mathcal{I}'$ be a solution to the MyCoPS problem. Let I' be a solution to the instance of 01KP as defined under the defined mapping and $\mathcal{J}'$ be the solution to the instance of MyCoPS under the defined mapping. The proof follows the previous one in structure.

$$\sum_{p_i \in P'} \textsc{SizeP}(p_i) \leq \Delta \implies \sum_{i \in I'} w_i \leq W \,,$$

under the identified mapping. Also,

$$\sum_{p_i \in P'} \textsc{Utility}(p_i) \geq \mathcal{V} \implies \sum_{i \in I'} z_i \geq Z \,.$$

Hence I' is a valid set. □

Theorem 3 (MyCoPS and NP Completeness)**.** *MyCoPS is NP-complete.*

Proof. This follows trivially from Lemmas 1 and 2. □

6. Partitioned Memory Model for Profile Management

Since profile management has been shown to be an intractable problem in general, one possibility of conquering the complexity is to find special cases where the complexity of profile management is polynomial. Another is to use heuristics that can provide good enough solutions. We will investigate both in the remainder of this paper, where this section focuses on a special case and Sections 7, 8 and 9 describe and evaluate heuristics that solve this problem.

As previously mentioned, when agents interact they will store information about these interactions. This information is contained in a profile which may have empty items even after multiple interactions. Missing information will lead to a lower utility as memory is consumed but contains no useful information. However, the profile may still need to be kept in memory. One way to increase the utility of the contents of information in memory is to ensure that empty items are not held in memory. Therefore, we *partition* (hence the partitioned memory model) the profiles and keep items of one specific type in a dedicated memory segment for that type of information. This aligns with techniques that memory-constrained agents use to *statically* allocate fixed-sized memory regions that contain items of the same type and size in a given memory region. In summary, rather than allocating a complete profile on one specific agent, we instead split the profiles into the individual items that make up the profile and store items of the same type in their own separate memory region.

A trust model running on such resource-constrained devices will use each of these regions to store one type of information relevant to the trust model, e.g., agents in the network, services offered by other agents, digital certificates, and others. Each region is assigned a fixed size at compile time to avoid memory fragmentation and so they all fit within the RAM of the agent.

We divide an agent n's memory M^n into R memory regions $\{M^n_1, \ldots, M^n_R\}$. Each region M^n_k contains a set of homogeneous items that represents data about one type of information that contributes to trust evaluation. Each region M^n_k has a size in bytes denoted by $\textsc{SizeR}(M^n_k)$. Each region M^n_k can contain zero or more items of the same type, where the j^{th} item of this memory region is denoted $I^{n,k}_j$. Each item in M^n_k has the same size in bytes given by $\textsc{SizeI}(M^n_k)$ due to them being homogeneous. Thus, the maximum number of items that a region M^n_k can contain is given by:

$$\left\lfloor \frac{\textsc{SizeR}(M^n_k)}{\textsc{SizeI}(M^n_k)} \right\rfloor . \tag{1}$$

All the information that agent n knows about agent m, is called a profile. It comprises all the unique types of information relating to m. As mentioned before, we assume a function called $\textsc{Subjects}(I^{n,k}_j)$, which returns the set of identities of the agents that a given item $I^{n,k}_j$ provides information on.

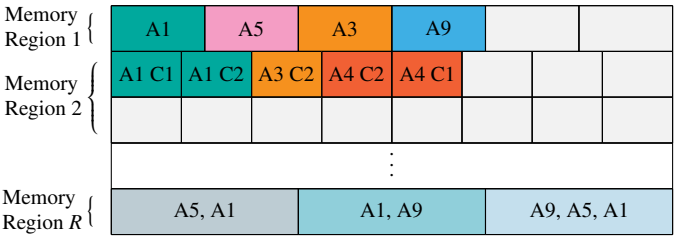


Figure 2: Example agent memory labelled with the target(s) of the information held in a partitioned memory model.
Information on different agents is represented as Ax, information on multiple agents is represented as a list (e.g., Aa, Ab, Ac), and information on the services provided by Ax is represented as Ax Cy.

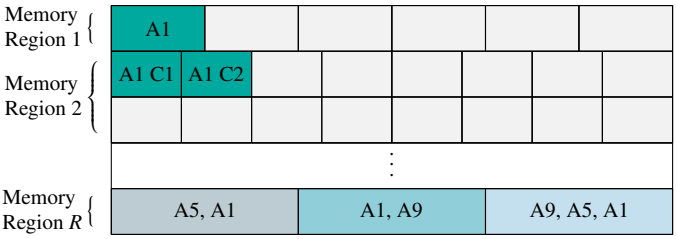


Figure 3: Example complete profile describing agent A1, with information on A1's two services C1 and C2.
Information on different agents is represented as Ax, information on multiple agents is represented as a list (e.g., Aa, Ab, Ac), and information on the services provided by Ax is represented as Ax Cy.

Definition 6.1 (Memory Slice)**.** Given an agent n and its memory $M^n = \{M^n_1, \ldots, M^n_R\}$, a memory slice (also called a slice) of agent n is a set of sets of items, one from each memory region, i.e.,

$$\overline{M^n} = \left\{ k \mapsto s \mid \forall k, 1 \leq k \leq R \wedge s \subseteq M^n_k \right\} \tag{2}$$

Since the profile memory is now partitioned, we now define a partitioned profile.

Definition 6.2 (Partitioned Profile)**.** A *partitioned profile* of agent m stored in M^n is a set of R item sets where each item set is a subset of a unique memory region and each item in each item set is about agent m.

$$p^n_m = \left\{ k \mapsto \left\{ I^{n,k}_j \mid \forall j \cdot m \in \textsc{Subjects}(I^{n,k}_j) \right\} \mid \forall k, 1 \leq k \leq R \right\} \tag{3}$$

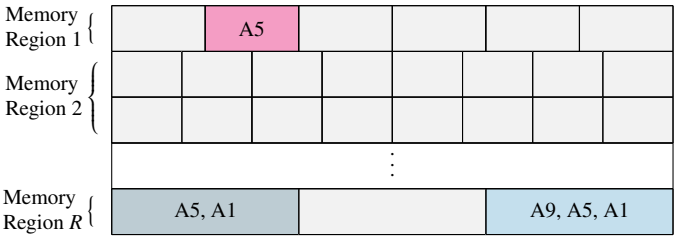


Figure 4: Example profile describing agent A5.
Information on different agents is represented as Ax, information on multiple agents is represented as a list (e.g., Aa, Ab, Ac), and information on the services provided by Ax is represented as Ax Cy.

A partitioned profile about agent m is a slice of agent n's memory that is predicated on m being a subject of items contained in a memory region. We denote by $p^n_m[k]$, the set of items of type k of that agent n holds about agent m. The number of sets contained with a partitioned profile (denoted R) is the same as the length of a profile (denoted l) as defined in Definition 4.3.

As before, an agent n may not have all information about another agent m (i.e., an incomplete profile). We say that agent n has a profile for agent m if $\exists k, 1 \leq k \leq R, p^n_m[k] \neq \emptyset$. Likewise, an agent n has a complete profile of agent m if n has all required information about m, i.e., $\forall k, 1 \leq k \leq R, p^n_m[k] \neq \emptyset$.

Similarly, we still assume that some items held by agent n about m are vital for interactions to occur (e.g., cryptographic information) and are called *crucial items*. In this paper, we assume that there always exists a memory region that holds the vital items and without loss of generality we assume that these vital items are held in the first region in memory, i.e., in M^n_1.

An example agent memory is shown in Figure 2 with multiple memory regions each divided into items of different sizes. Items coloured in light grey are empty and contain no information. Example memory regions are included to cover the different kinds of subjects memory items contain. Either, (i) an item about a single agent, (ii) an item about a single agent and a single service it provides, and (iii) an item containing a set of information about multiple agents. A complete profile for the agent A1 is shown in Figure 3 where items exist in all the memory regions. An incomplete profile for A5 is shown in Figure 4 as memory region 2 contains no items that A5 is the subject of.

6.1. Updating Trust Model Storage

As in the unified memory model, the execution of an action by an agent can cause an update of the memory state at one or more agents. For example, an agent may disseminate the services it provides and this information is updated on all agents that subscribe to announcements about that service. For these definitions, the state of a memory region before the execution of an action is denoted by M^n_i and the state after the execution of the action by $\bigcirc M^n_i$. We revisit the previous definitions and define them for the partitioned memory model.

Definition 6.3 (Insertion)**.** Given an agent n with memory $M^n = \{M^n_1, \ldots, M^n_R\}$. We say that item e is inserted into a memory region $M^n_i \in M^n$ if $e \notin M^n_i \wedge \bigcirc M^n_i = (M^n_i \cup \{e\})$, and the type of e is the same as M^n_i.

Definition 6.4 (Removal)**.** Given an agent n with memory $M^n = \{M^n_1, \ldots, M^n_R\}$. We say that item e is removed from a memory region M^n_i if $e \in M^n_i \wedge \bigcirc M^n_i = (M^n_i \setminus \{e\})$.

Definition 6.5 (Eviction)**.** Given an agent n with memory $M^n = \{M^n_1, \ldots, M^n_R\}$. We say that *$r$ is evicted for item e* in a full memory region M^n_i if r is removed from M^n_i and e is inserted into M^n_i, i.e., $r \in M^n_i \wedge e \notin M^n_i \wedge \bigcirc M^n_i = ((M^n_i \setminus \{r\}) \cup \{e\})$.

When such an item e exists, we will only say that item r is evicted. We also make the assumption that an eviction is always performed when new information becomes available and the relevant memory region is full.

In the partitioned memory model, eviction performs differently to the original memory model. In the original memory model an eviction of information on one agent, will cause all information on that agent to be evicted. However, in the partitioned model, eviction of one item will only lead to that item being removed and all other items who share the subject with the item will remain in memory.

7. Memory and Eviction Strategies

In this section we describe an example instantiation of memory regions that are used to store information used by a trust model and the eviction strategies investigated. We also propose a method to evaluate the *utility* of information in memory to gauge the performance of the eviction strategies.

7.1. Example Memory Usage

In this system, trust models are built from past interactions to enable agents to make decisions about whom to perform future interactions with. The relevant information held in four memory regions, each containing different types of information that are directly relevant to assessing trust. The set of memory regions that exists on an agent n is $M^n = \{M^n_C, M^n_\mathcal{T}, M^n_\mathcal{R}, M^n_\mathcal{S}\}$. Each of these regions exists as a set of items, however, as these items will refer either to an agent or a service an agent provides, we represent them as functions to simplify their use.

- M^n_C A database of cryptographic information (such as shared secrets) used to securely send information between a trustor and trustee. We assume that this information also contains the details on the services supported by the agent. This is represented as $M^n_C : V \to \mathbb{B}$, indicating if an agent has cryptographic information for a specific agent.
- $M^n_\mathcal{T}$ A database about directly observed interactions with the trustees. This is represented as $M^n_\mathcal{T} : V \times S \to (\Gamma \cup \{\perp\})$, where Γ is the representation of the trust model. For example, Γ would be $\mathbb{N}_0 \times \mathbb{N}_0$ for the BRS which counts the number of correct and incorrect interactions via a pair of integers. A trust model may equal $\perp$, which indicates the trust model contains no information or is missing.
- $M^n_\mathcal{R}$ A database of peer-provided reputation information on the trustees. This is represented as $M^n_\mathcal{R} : V \to (V \times S \to (\Gamma \cup \{\perp\}))$ which is the state of the direct trust disseminated by another agent.

$M^n_{\mathcal{S}}$ A database of stereotype information used to bootstrap trustworthiness evaluation early in the system lifetime or when new agents join the system. This is represented as $M^n_{\mathcal{S}} : V \times S \rightarrow \Gamma$.

7.2. Eviction Strategies

The following eviction strategies are explored in this paper. We divide these strategies into those that focus on a single memory region, those that focus on a single memory region and use output from trust evaluation, and those that consider the contents of multiple memory regions.

7.2.1. Single Memory Region Eviction Strategies

These eviction strategies evaluate which item to evict in a single memory region for an item of the same type as is stored in the memory region.

- None — No items are ever evicted.
- Random — A random item is evicted.
- FIFO — The oldest item is evicted in a first-in first-out order.
- LRU — The least recently used item is evicted.
- LRU-2 — The item whose penultimate use time is the smallest is evicted.
- MRU — The most recently used item is evicted.

The age of an item for FIFO is determined by the time at which the item was added to the memory region. The use time of an item for LRU, LRU-2 and MRU is set to the time at which a task is offloaded to the agent an item is the subject of. The use time is also updated when cryptographic information is used to receive a message.

7.2.2. Single Memory Region with Trust Eviction Strategies

There are two eviction strategies that need trust models to evaluate a measure of trust. Both of these strategies only consider an eviction for the contents of $M^n_{\mathcal{T}}$. All other memory regions use LRU to decide which item to evict.

- Chen2016 [23] — The paper describes this as "pop out the trust value towards the earliest interacting node among those with trust values below the median" which we interpret as using LRU on trust values less than the median. However, the paper makes no mention of what action to take if there are no values lower than the median. In this case we perform LRU on all items.
- FiveBand — Partition the items of the trust memory region into 5 quantiles of equal size. Use LRU on items with a trust values between 20% – 40% and 60% – 80%. If there are no values within these bands, use LRU on all items. The intuition is that it is preferable to keep information on new agents (40% – 60%), good agents (80% – 100%), and bad agents (0% – 20%).

7.2.3. Priority-based Eviction Strategies

We propose one eviction strategy that considers that different services may have different priorities. These service priorities can be provided by the designer of a system, when multiple services are present and some are critical and others are non-critical. Prioritising which information to keep aims to ensure critical services have a higher chance of success.

- CapPri — Perform LRU on items that contain information about a service with the same lowest priority.

7.2.4. Multiple Memory Region Eviction Strategies

We propose two eviction strategies that consider the contents of multiple memory regions. The intuition is that better decisions can be made about what to keep if eviction is not considered locally to a single memory region, but also takes into account what relevant information is stored in other memory regions.

- NotInOther — Find all items in a memory region with no corresponding entry in other regions (i.e., not the same subject), use LRU for eviction. If no item matches this criteria, use LRU on all items.
- MinNotInOther — Find all items in a memory region with the smallest number of regions with a corresponding entry, use LRU to choose which to evict. If no items match this criteria, use LRU on all items.

For NotInOther, an example would be if a stereotype is known for one agent but no other information is kept on that agent in any other memory regions, it provides low utility and should be evicted. MinNotInOther takes this further and instead find items that have the smallest number of related items in other buffers. So, if a stereotype and cryptographic information is known for one agent and another agent has information in each buffer, then an item belonging to the first agent would be evicted. This aims to keep a broad amount of information on selected agents in memory.

7.3. Interaction Selection Strategy

The behaviour of eviction strategies can depend on the selection strategy of a trust model. For example, when an eviction strategy requires a trust value (e.g., evicting information on the least trusted agents), or is based on the time that information was last used (a higher trust value implies more frequent use). This means that it is necessary to use a trust model as part of these experiments. We employ the BRS to evaluate trust in this case as it is a very small model which can be effectively used on the resource-constrained devices with limited RAM that this work is concerned with. The agent with which an interaction will be performed is selected randomly from the agents who have a trust value within 0.1 of the maximum trust value (with trust being measured between 0 and 1).

7.4. Utility of Data in Memory

In order to quantify the usefulness of the data held we propose a trust model agnostic utility function. It quantifies if information is present in memory for agents that behave well

for an interaction between two agent's services. This occurs irrespective of the trust model in use. The intuition is that if a trust model held by an agent has more information on agents who perform interactions correctly, then that agent is more likely to select a good agent to interact with.

If an agent n is unaware of another agent m's cryptographic information they will be unable to communicate, so n will not select m for an interaction. This has been reflected in the utility function by n not deriving any utility for information about m if m's cryptographic information is not known.

The utility of the data in the memory held on agent n for a task t generated by service s is calculated by Equation 4. A utility for each memory region is calculated and then weighted based on the expected importance to the trust model.

$$U(s,n,t) = \frac{\sum\limits_{n' \in V'_{s,n,t}} U_C(n,n',s) \sum\limits_{r \in M^n} w_r \, U_r(n,n',s)}{|V'_{s,n,t}|} \tag{4}$$

Where:

- $V'_{s,n,t}$ is the set of agents except agent n with the matching service s that will perform well (identified by an oracle function $\textsc{Good}(s,v,t)$ for the task t). These are agents that it would be good to interact with.

$$V'_{s,n,t} = \{\, v \mid v \in V \setminus \{n\} \wedge s \in S_v \wedge \textsc{Good}(s,v,t) \,\} \tag{5}$$

- w_r is a per-region weight that specifies the relative importance of different types of information. The weights must sum to one, $1 = \sum_{r \in M^n} w_r$.

- $U_C(n,n',s)$ determines if agent n has the cryptographic information to communicate with agent n'.

$$U_C(n,n',s) = \begin{cases} 1 & \text{if } M^n_C(n') \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

- $U_{\mathcal{T}}(n,n',s)$ determines if agent n has previously recorded an interaction with service s provided by agent n'. It is important that this is not an empty trust model.

$$U_{\mathcal{T}}(n,n',s) = \begin{cases} 1 & \text{if } M^n_{\mathcal{T}}(n',s) \neq \bot \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

- $U_{\mathcal{R}}(n,n',s)$ determines if agent n has knowledge of peer-provided reputation information about service s provided by agent n'. This looks for an agent n'' that has disseminated reputation information about agent n'.

$$U_{\mathcal{R}}(n,n',s) = \begin{cases} 1 & \text{if } \exists n'' \in V, M^n_{\mathcal{R}}(n'')(n',s) \neq \bot \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

- $U_{\mathcal{S}}(n,n',s)$ determines if agent n knows stereotype information for service s provided by agent n'.

$$U_{\mathcal{S}}(n,n',s) = \begin{cases} 1 & \text{if } (n',s) \in M^n_{\mathcal{S}} \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

In these experiments each of the data types are weighted equally. However, for trust model specific evaluations, different weights could be provided based on an understanding of the relative importance of different information to the trust model.

8. Experimental Setup

As selecting which items to keep is NP-complete, there is a need to investigate suitable eviction heuristics. We now describe the experimental setup used to perform simulations which investigate the effectiveness of different eviction strategies. These simulations were performed with a custom discrete event simulator[2]. The implementation of the eviction strategies can be found in this source code. There are 10 agents ($|V| = 10$) each with 2 services ($|S| = 2$), the utility weights are set to be equal for each type of information ($w_b = 1/|M^n|$), and other parameters of simulation are shown in Table 1. Over 60 repeats were performed for each parameter combination.

Each agent in the system instantiates their memory with regions described in Section 7.1 and the sizes shown in Table 2. The capacity shows how much of the total amount of information could possibly be stored. For the 10 agents simulated there are a maximum of 9 M^n_C entries, 18 $M^n_{\mathcal{T}}$ entries, 9 $M^n_{\mathcal{R}}$ entries and 18 $M^n_{\mathcal{S}}$ entries.

In these experiments the memory sizes and eviction strategies are varied in order to focus on investigating the relation between the memory allocated to different types of data, the management of that information, and other factors such as the size of the network. This reflects the resource-constraints which information management needs to be performed under due to the low memory (32 to 256 KiB) available in the target devices.

8.1. Actions

There are three actions that agents perform:

1. Service Dissemination: Agents in the system announce their services to the rest of the network. This can be via a publish-subscribe protocol such as MQTT [32].
2. Cryptographic Information Dissemination: When agents encounter another agent that they do not have the necessary cryptographic information to communicate with, they can request it from a certificate authority.
3. Stereotype Request: Agents may request stereotypes about other agents and their services. Stereotypes are used to bootstrap trust evaluation when the system is new, or there is little information about another agent.

There are a two periodic actions that agents perform:

1. Trust Dissemination: Agents will periodically disseminate their trust values to all other agents which then consider storing this as reputation information. This dissemination occurs on average every $\mathcal{P}_{\text{dissem}}$ seconds, which is modelled using the exponential distribution $\text{Exp}(\mathcal{P}_{\text{dissem}})$.

[2] Available at `https://github.com/MBradbury/trust-buffer-simulation`.

Table 1: Simulation Parameters

Name	Value	Description
Trust Dissem. Period	$\mathcal{P}_{\text{dissem}} = 1$ s	The average time between an agent disseminating its trust values.
Interaction Period	$\mathcal{P}_{\text{interact}} = 1$ s	The average time between an agent's service performing an interaction.
Duration	300 s	The time before the simulation halts.
Start Delay	1 s	The maximum random start delay.
Interaction Strategy	BRS/Banded	How agents choose who to perform an interaction with.

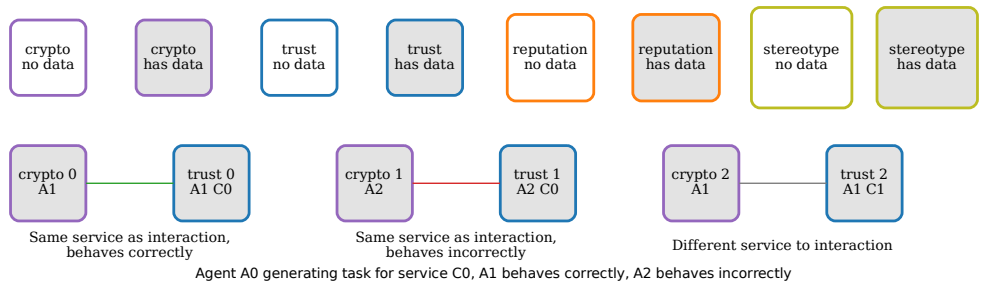


Figure 5: Legend showing elements of Figures 6, 7 and 8. The top row shows items in memory regions that do (grey background) or do not (white background) contain information. The bottom row shows example relations between items in the crypto and trust memory regions. The number after the name of memory region is the index into that memory region at which that item is stored. The information after the name and index is the subject of the item. For example "A1" means the subject is agent A1 and "A1 C0" means the subject is agent A1's service C0.

Table 2: Simulation Memory Sizes

Name	$\lvert M^n_C\rvert$	$\lvert M^n_{\mathcal{T}}\rvert$	$\lvert M^n_{\mathcal{R}}\rvert$	$\lvert M^n_S\rvert$	Capacity (%)
Complete	10	20	10	20	100
Large	10	10	10	10	78
Medium	10	5	5	5	53
Medium2	5	10	5	5	49
Medium3	5	5	10	5	53
Medium4	5	5	5	10	49
Small	5	5	5	5	42

2. Task Interaction: Agents will periodically generate a task which requires an interaction with another agent who provides the same service the task was generated by. This occurs on average every $\mathcal{P}_{\text{interact}}$, which is modelled using the exponential distribution $\text{Exp}(\mathcal{P}_{\text{interact}})$.

8.2. *Services Profiles*

We model the behaviour of each service s of each agent n as a hidden Markov model (HMM) $\lambda_{n,s} = (Q, O, \pi, A, B)$, where Q is a finite set of states, $\pi : Q \to [0, 1]$ is an initial probability distribution over the states in Q, $A : Q \times Q \to [0, 1]$ is the state transition matrix, O is a finite set of potential observations, and $B : Q \times O \to [0, 1]$ is the observation probability matrix. We define every $\lambda_{n,s}$ to have two hidden states $Q = \{\text{good behaviour (G)}, \text{bad behaviour (B)}\}$ and two observations that can arise from interactions with n, $O = \{\text{correct (✓)}, \text{incorrect (✗)}\}$. Four service profiles are defined in Table 3. When a simulation uses one of these profiles, it always includes 2 agents where both services are always bad and the remaining 8 agents are the specified service behaviour.

Table 3: Agent Service Profiles

Name	π	A			B		
			G	B		✓	✗
Always Good	$\begin{bmatrix}1.0\\0.0\end{bmatrix}$	G B	$\begin{bmatrix}1.0 & 0.0\\0.0 & 1.0\end{bmatrix}$		G B	$\begin{bmatrix}1.0 & 0.0\\0.0 & 1.0\end{bmatrix}$	
			G	B		✓	✗
Always Bad	$\begin{bmatrix}0.0\\1.0\end{bmatrix}$	G B	$\begin{bmatrix}1.0 & 0.0\\0.0 & 1.0\end{bmatrix}$		G B	$\begin{bmatrix}1.0 & 0.0\\0.0 & 1.0\end{bmatrix}$	
			G	B		✓	✗
Very Good	$\begin{bmatrix}0.99\\0.01\end{bmatrix}$	G B	$\begin{bmatrix}0.99 & 0.01\\0.80 & 0.20\end{bmatrix}$		G B	$\begin{bmatrix}0.99 & 0.01\\0.0 & 1.0\end{bmatrix}$	
			G	B		✓	✗
Unstable	$\begin{bmatrix}0.5\\0.5\end{bmatrix}$	G B	$\begin{bmatrix}0.5 & 0.5\\0.5 & 0.5\end{bmatrix}$		G B	$\begin{bmatrix}0.9 & 0.1\\0.0 & 1.0\end{bmatrix}$	

9. Results

To highlight the state of memory regions when calculating the utility, we first show three visualisations of memory contents when an agent's service needs to perform an interaction. Figure 6 shows the state of memory early in execution when there has been little opportunity to gather evidence. Figures 7 and 8 both show memory state late into information gathering when all agents behave correctly and some agent services behave incorrectly respectively.

In these graphs each box represents an item in a memory region. A line between boxes represents that those item store information about the same agent and potentially the service that the agent provides. Each graph shows the memory state

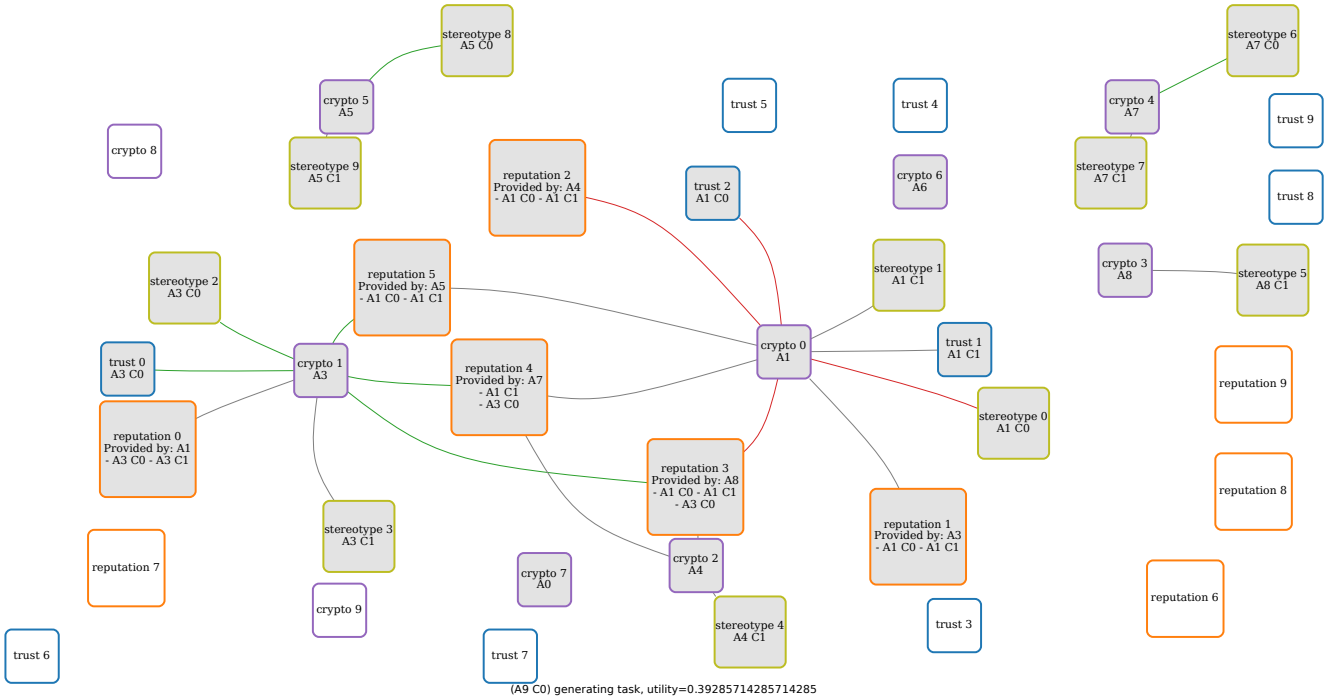


Figure 6: Memory contents for agent 9 early in the simulation when needing to interact with another agent's service 0.
Nodes are items stored in memory, different item types have different coloured bordered and when the background is grey it indicates that a value is held in that item. Edges between nodes indicate that they refer to the same agent. Green edges are for agent services that behave correctly and red edges are for agent services that behave incorrectly. Grey edges are for information that is about the same agent, but a different service that is provided.

for the agent that needs to perform an interaction. Grey lines between boxes indicates relations for the service that is not being executed, green lines indicate that information is held on an agent and service that will perform correctly and red lines indicate that information is held on an agent and service that will perform incorrectly. This information is shown in Figure 5.

The low utility in Figure 6 is due to the under-use of storage, whereas the lower utility in Figure 8 compared to Figure 7 occurs due to the memory space allocated to agents and services that perform poorly or for different services other than the one performing an interaction. This demonstrates (i) the importance of making use of available memory and (ii) the importance of ensuring useful information is stored in memory.

9.1. Always Good Behaviour

The evolution of the normalised utility (utility divided by maximum utility) for a single run when 8 agents are always good and 2 are always bad is shown in Figure 9. There is an initial period where the utility starts low and then rapidly increases. This is because initially no information is known, and as agents begin interacting information is recorded in memory. The utility does not decrease as the memory regions are large enough for information on all agents and their services to be stored, so there are never any evictions.

9.2. Varying Behaviour and Memory Region Sizes

The graphs in Figures 10, 11, 12 and 13 show the normalised utility of eviction strategies when varying agent behaviour and memory region sizes. These graphs show boxplots where the central bar is the median, the dot is the mean, the box starts at the first quartile and ends at the third quartile, and the upper and lower whiskers extend to the first item less than and greater than 1.5 times the inter-quartile range respectively. Outliers are not shown as there are a large number due to the initial period of information gathering. Results been sorted by the median.

9.2.1. No Eviction

Figure 10 shows that when memory regions are large enough to store all information, every eviction strategy performs exactly the same because no eviction is required. However, the normalised utility is lower for agents whose behaviour is worse, even though there is space to store all data. The interaction selection algorithm has an effect on information utility because some agents may not be chosen to perform an interaction with. This means no records of direct trust for agents not selected for interaction and it then impacts other agents who would not learn of those interactions via reputation.

9.2.2. Evictions

Most eviction strategies have similar normalised utility, except MRU, None and CapPri which tend to be the three worst performing strategies. The worst-case difference between the best and worse performing eviction strategy is 25 percentage points and the average distance is 13 percentage points, indicating similar normalised utility between eviction strategies.

MRU tends to perform poorly as it prioritises evicting the most recently used information. This means when an agent's

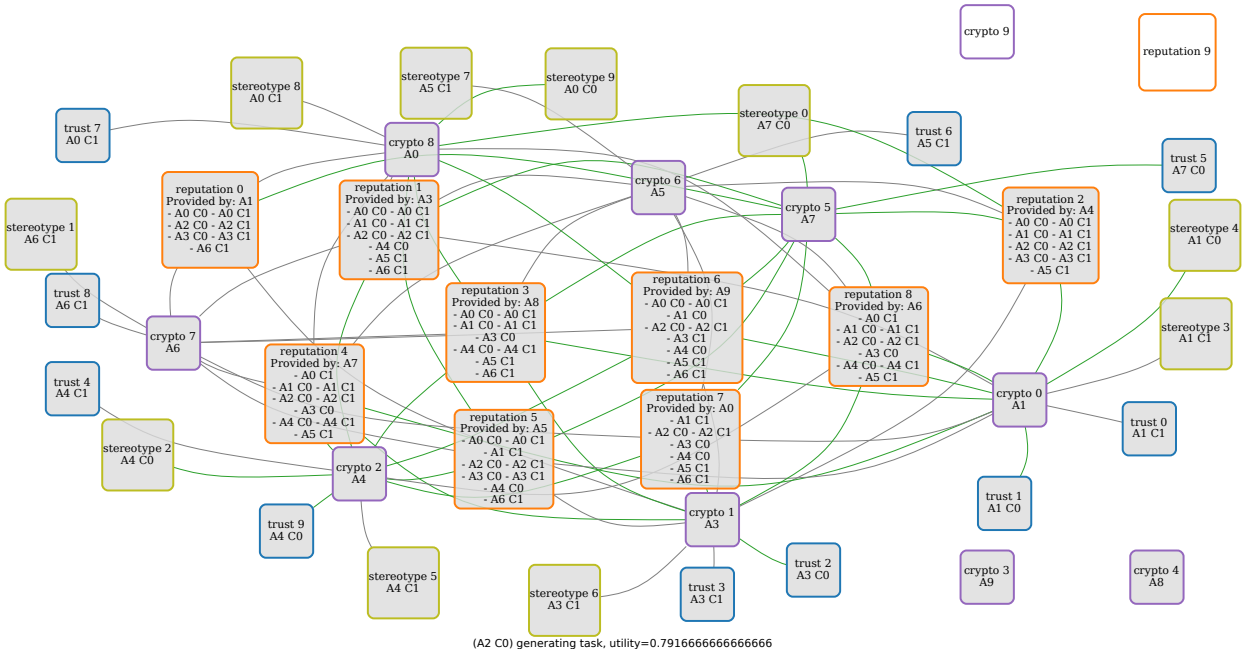


Figure 7: Memory contents for agent 2 late in the simulation when needing to interact with another agent's service 0 and all agents would perform correctly. Nodes are items stored in memory, different item types have different coloured bordered and when the background is grey it indicates that a value is held in that item. Edges between nodes indicate that they refer to the same agent. Green edges are for agent services that behave correctly and red edges are for agent services that behave incorrectly. Grey edges are for information that is about the same agent, but a different service that is provided.

service is selected for an interaction because it is believed to be good, that information will be removed next.

CapPri tends to perform poorly because it optimises for keeping information on high priority services, which is different to the utility function. This causes CapPri to have the largest interquartile range because service C1 has a higher priority than service C0 as demonstrated in Figure 14.

The eviction strategy to never evict (None) performs poorly except when the memory size is small. When items are stored, due to the ordering of events, items related to the same agent and service will be added to different memory regions at similar times. Without eviction, any badly behaving agent services cannot be removed. No eviction performs fairly well when the buffers are small because when agents perform well (Very Good and Good) and poorly when agent's services are less reliable (Unstable). This is because there is a higher chance that information on well behaved agents will be stored for Very Good and Good, but lower for Unstable. This occurs even though the simulation randomises the start times of agents.

FIFO, LRU, LRU2, Random, Chen2016 and FiveBand and NotInOther all have similar normalised utility. For Chen2016, FiveBand and NotInOther this can be attributed to them using LRU on a subset of the information stored. Why strategies that use LRU perform similar to FIFO and Random relates to the data access pattern, which is linked to performing an interaction with an agent where the it is most likely to succeed. There will be different access patterns depending on a variety of factors (e.g., multiple memory regions, memory sizes, agent behaviour, trust model). Future work will be needed to understand the performance in this different scenario (e.g., [33]).

9.3. Comparing Different Allocations with the Same Capacity

MinNotInOther tends to perform best when an eviction of cryptographic data M_C^n would have a large impact on the utility (large and small buffer sizes in Figures 11 and 13), but does not perform as well when evictions of other data would have a large impact (medium memory size in Figure 12). This can also be observed with the different sized medium memory in Figure 15. Only when there is an excess of space allocated to M_C^n does MinNotInOther perform worse than techniques other than MRU, None and CapPri. By having an excess of space in M_C^n, it means that no item in the other three buffers will have their utility discounted, so a local optimisation on each buffer will work well by keeping the most frequently used without eliminating any options. Whereas, when there is not an excess of space, MinNotInOther is able to use knowledge of what is in each buffer to prioritise keeping it.

9.4. Eviction Strategy Memory Overhead

As there is limited amount of memory, an eviction strategy needs to avoid imposing a high memory overhead. Selecting an eviction strategy needs to consider its performance and also the trade-off that needs to be made between the information needed by the eviction strategy and the trust model.

The memory overheads for the investigated eviction strategies are relatively low. For no eviction and FIFO there is no memory cost, for Random there is a fixed cost for the PRNG state, for LRU-k there is a cost to store k counters, and for all others one counter needs to be stored. While the memory cost may be low, when information has a small representation, e.g., 8 B, adding a single 4 B counter will lead to a 50% increase in size. So when

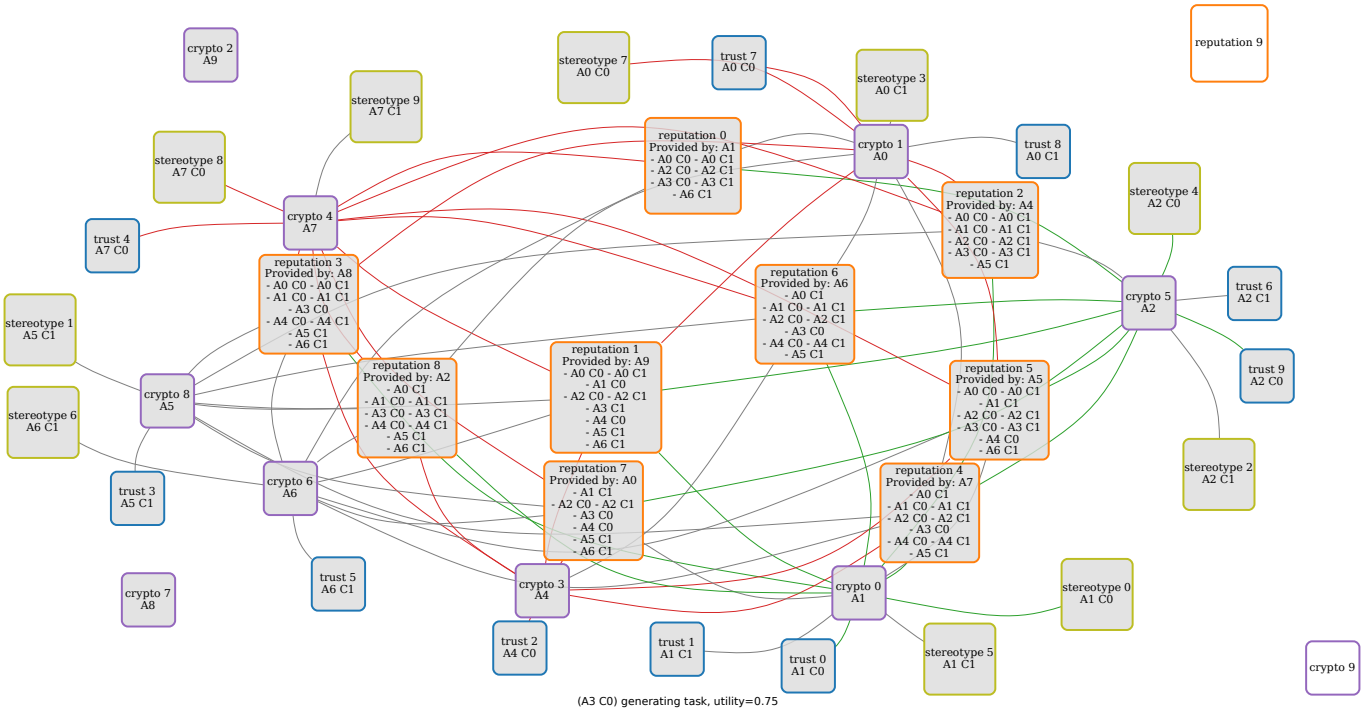


Figure 8: Memory contents for agent 3 late in the simulation when needing to interact with another agent's service 0 and agents 0, 4, and 7 would perform incorrectly. Nodes are items stored in memory, different item types have different coloured bordered and when the background is grey it indicates that a value is held in that item. Edges between nodes indicate that they refer to the same agent. Green edges are for agent services that behave correctly and red edges are for agent services that behave incorrectly. Grey edges are for information that is about the same agent, but a different service that is provided.

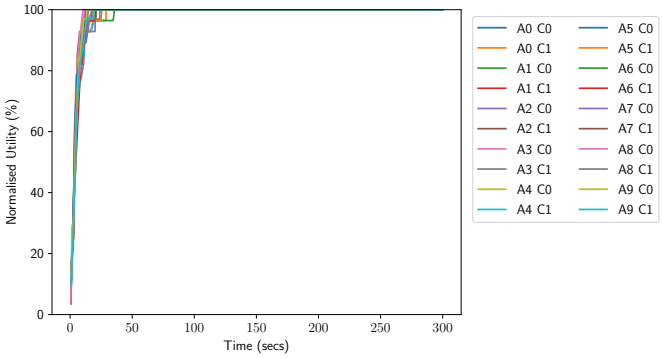


Figure 9: Utility over time for AlwaysGood with no evictions.

choosing how large to make each memory region, the eviction strategy memory cost needs to be considered.

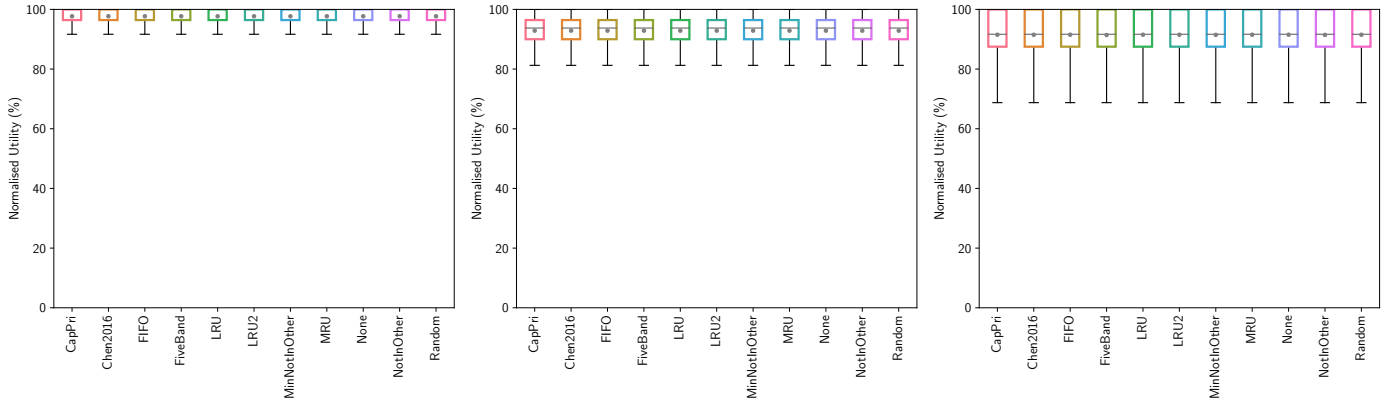
9.5. *Eviction Strategy Summary*

So the conclusions to drawn from these results are that it is important to understand how memory will be allocated between different memory regions with respect to the expected number of agents and services in the network. Different combinations of agent behaviours and memory capacity will require different eviction strategies. Overall, MinNotInother, NotInOther, Chen2016, FiveBand, FIFO, LRU, LRU2 and Random performed well with a maximum of 10 percentage points between their medians in these simulations. MRU and None should be avoided due to their worse ability to provide a high utility. In the worst case an agent will need to process much more information than it can store, in which case MinNotInOther would be the best eviction strategy according to this utility function.

10. Sizing Memory Regions

We now address the problem of how to divide up the available finite memory into different memory regions that will be statically allocated at compile time. This can be mapped to the Bounded Knapsack Problem [34] where the *weight* of the items to store corresponds to their size in memory (w_r) and the maximum number of items ($\widehat{x_r}$) corresponds to the point where no more utility is gained by having more of that type of item. This modelling is performed to obtain results for the variable x_r which is the size of the memory region r. Bounds on the maximum number of items from which value can be obtained is specified using u_r. In this scenario, u_r specifies either either the number of agents or the total number of agent-provided services depending on the memory region r. In this section we focus on the optimisation problem as opposed to the decision problem presented earlier.

We assume that an upper bound on the expected number of agents is known so the model remains linear. The size of a reputation item w_R could be specified in terms of the number of agents in the system x_C, but as this would make the model quadratic, the size is a function of the upper bound of the number of agents. We also assume the number of services is known at compile time as an agent will only store information on services that it uses and/or provides.



(a) Agent behaviour: Very Good

(b) Agent behaviour: Good

(c) Agent behaviour: Unstable

Figure 10: Memory size: Complete

This problem can be formulated as the following objective optimisation problem for agent n to find values for x_r:

$$\begin{aligned} &\text{maximise} && \sum_{r \in M^n} v_r \, u_r \\ &\text{subject to} && \sum_{r \in M^n} w_r \, x_r \leq W \\ & && \forall r \in M^n, x_r \in \mathbb{N}_0 \wedge 0 \leq x_r \leq \widehat{x_r} \end{aligned} \tag{10}$$

For the four buffers we are focusing on in this work we set the following upper bounds and per-buffer region functions:

$$\begin{aligned} \widehat{x_C} &= |V| - 1 & u_C &= \min\{\, x_C, |V| - 1 \,\} \\ \widehat{x_\mathcal{T}} &= |S|\,(|V| - 1) & u_\mathcal{T} &= \min\{\, x_\mathcal{T}, |S|\, u_C \,\} \\ \widehat{x_\mathcal{R}} &= |V| - 1 & u_\mathcal{R} &= \min\{\, x_\mathcal{R}, u_C \,\} \\ \widehat{x_\mathcal{S}} &= |S|\,(|V| - 1) & u_\mathcal{S} &= \min\{\, x_\mathcal{S}, |S|\, u_C \,\} \end{aligned}$$

These constants are set such that:

- An additional item in a buffer provides no utility if there is already enough space for all possible items.
- The size of the M^n_C buffer impacts the maximum size where other buffers still provide utility.
- The item's weight $w_\mathcal{X}$ is its size in bytes.
- W is the maximum amount of memory that can be used.
- The value $v_\mathcal{X}$ is an indication from a trust model on the relative importance of different information.

This optimisation problem is implemented using IBM ILOG CPLEX version 12.8. Results of the model are shown in Table 4 where $\hat{W}$ is the total amount of memory used and Cap is the percentage of items that the combination of buffer sizes x is capable of holding. The number of agents $|V|$ is set to 10 and each agent has their number of services $|S|$ set to 2. We chose the following sizes of buffer items (in bytes):

- $w_C = 160$ based on secp256r1 keys and signatures [35].
- $w_\mathcal{T} = w_\mathcal{S} = 8$ based on BRS (two 4 B integer counters).
- $w_\mathcal{R} = |S|\,(|V| - 1)\, w_\mathcal{T}$ to store reputation information provided by peers. $w_\mathcal{R} = 144$ when $|V| = 10$ and $|S| = 2$.

In general, optimal solutions produced in Table 4 allocate memory to small items first before large items. This is different for M^n_C items, where memory needs to be allocated to these vital items first before space can be allocated to smaller items.

Depending on the trust model in use, different entries in different buffers may have unequal importance in evaluating a measure of trust. The last four rows of Table 4 show results when each buffer is valued higher than the others. A higher value for $M^n_\mathcal{T}$ and $M^n_\mathcal{S}$ leads to no changes, as these are the smallest items. Whereas increasing the value of M^n_C and $M^n_\mathcal{R}$ reduce the percentage of items that can be stored in order to allocate a greater amount of space to larger items.

When performing a deployment a user can specify these parameters in order to obtain an optimal set of buffer sizes. A downside is that this requires the user to know the number of agents in the system a priori and to value the relative importance of different information used by the trust model.

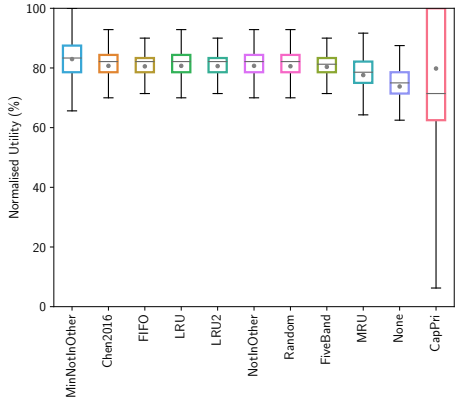
11. Discussion

11.1. Deciding To Evict

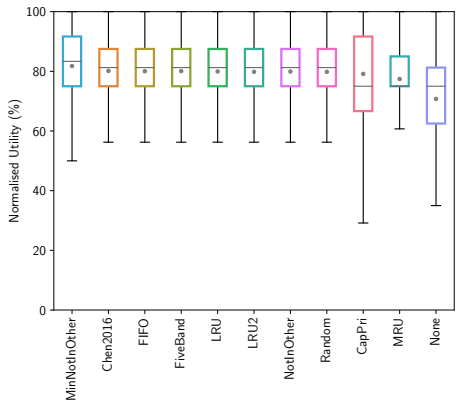
In this work an eviction is always performed when a memory region is full and an item needs to be added. Ideally, an agent would only evict and replace an item if doing so would increase the utility of the items in memory. However, an agent cannot necessarily implement this utility function because it requires an oracle that knows how all agent's services would behave for an interaction. We conjecture that the dual of the MyCoPS problem, selecting which nodes to evict, will be NP-complete.

11.2. Alternate Utility Functions

In this work we have assumed that trust-based interaction selection algorithms will be most effective if different types of information is stored about the same agents and their services

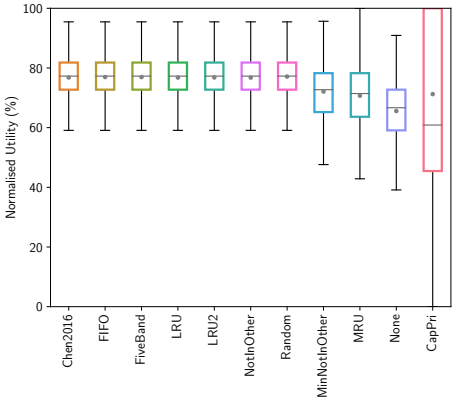


(a) Agent behaviour: Very Good

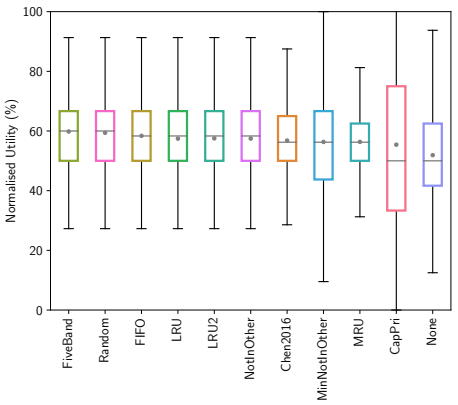


(b) Agent behaviour: Unstable

Figure 11: Memory size: Large

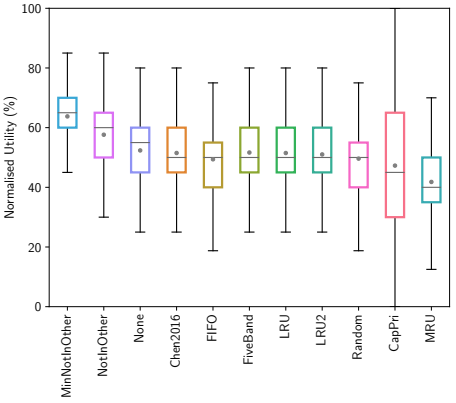


(a) Agent behaviour: Very Good

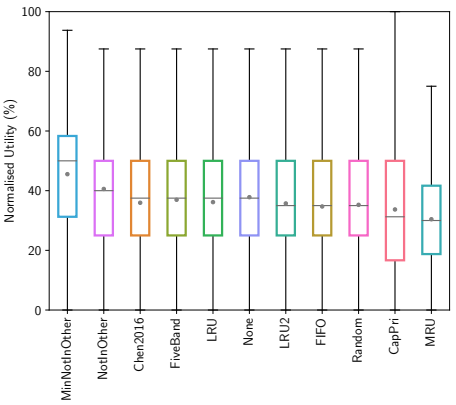


(b) Agent behaviour: Unstable

Figure 12: Memory size: Medium



(a) Agent behaviour: Very Good



(b) Agent behaviour: Unstable

Figure 13: Memory size: Small

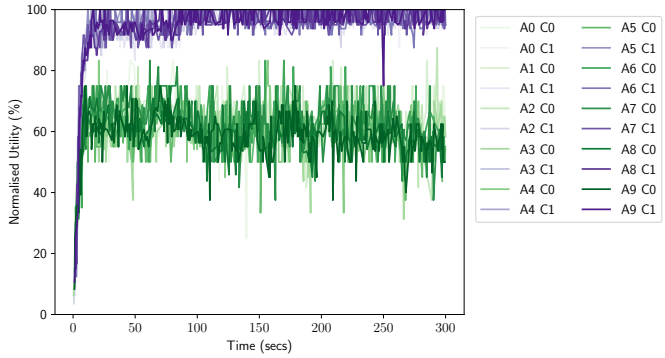


Figure 14: Divergence in utility for two services when behaviour is Good, buffers are large and CapPri is used to evict.

that behave well. However, not all selection algorithms will produce optimal outputs with this set of information; some selection algorithms may perform better if information is kept on the best and worst performing agents. Alternate approaches might calculate utility according to aspects such as age [36], usefulness of information, or other metrics.

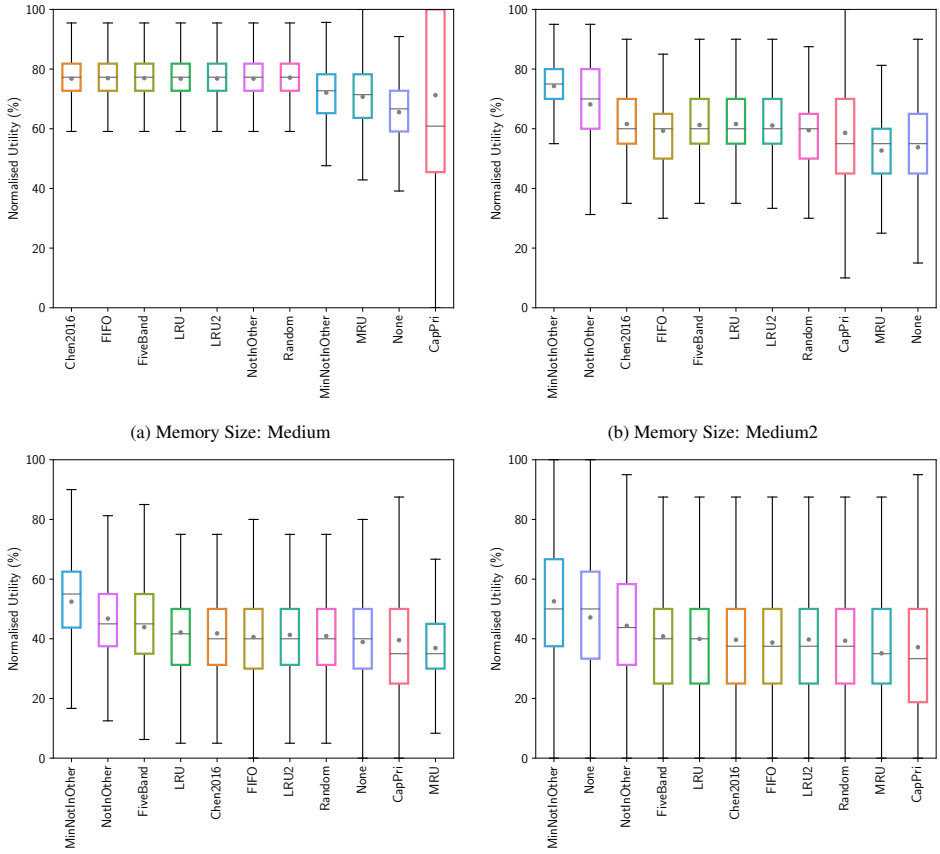
11.3. Assumption of Correctly Performed Actions

We assume that both the dissemination and request actions are performed correctly. There is little utility that could be gained by performing request incorrectly, as it prevents an agent from being able to request an interaction. However, performing disseminate incorrectly would allow agents to supply incorrect information to their peers. Specifically it could be beneficial for peers to lie about the contents of their trust memory region which other agents would store in the reputation memory region. We have chosen to not focus on this threat, due to the large amount of related work in this area [5, 37, 38]. Our approach is also flexible, so if a trust model needs to be deployed without reputation (and this threat) then the reputation memory region $M^n_{\mathcal{R}}$ can be omitted from the analysis.

12. Conclusions and Future Work

In this work we have investigated how to manage information stored on resource constrained agents that is used to assess behavioural trust in other agents. Work in cache replacement policies are not directly applicable, as deleted information used to evaluate trust cannot be recreated or re-obtained.

We showed that selecting which information to keep and/or to evict is NP-complete, so simulations of 11 different eviction



(a) Memory Size: Medium

(b) Memory Size: Medium2

(c) Memory Size: Medium3

(d) Memory Size: Medium4

Figure 15: Agent behaviour: Very Good

strategies were performed to gauge their performance. Results for our proposed trust model-agnostic utility function, showed that MinNotInother, NotInOther, Chen2016, FiveBand, FIFO, LRU, LRU2 and Random performed well. However, having the MinNotInOther eviction strategy that considers relevant items in other buffers can improve trust information utility when the memory size is small. Finally, we investigated how to size these memory regions under memory constraints via obtaining a solution to the Bounded Knapsack Problem.

Future work will need to consider heuristics to decide if an item should be evicted, taking into account both utility and the cost of re-adding items (e.g., verifying digital signatures [30]), plus other considerations such as data freshness in highly mobile networks. Another area for future work is considering how to specialise the utility function. In this work we have focused on evaluating utility per agent. Instead the utility function could consider the global balance between the information stored on agents, this may involve focusing on storing information on the best and worst performing agents, or by trading-off detail on some agents in order to store cryptographic information on more agents. Finally, distributed storage of trust information could be considered. However, this will incur additional storage costs to ensure the integrity, confidentiality and authenticity of behavioural trust information.

Acknowledgement

This work was supported by the PETRAS National Centre of Excellence for IoT Systems Cybersecurity [EPSRC Grant EP/S035362/1].

Data Statement

The software used to generate these results can be found at `https://github.com/MBradbury/trust-buffer-simulation`. The data gathered and presented in this paper can be found at `https://zenodo.org/record/4353611` [39].

References

[1] P. Ranaweera, A. D. Jurcut, M. Liyanage, Survey on Multi-Access Edge Computing Security and Privacy, IEEE Communications Surveys Tutorials 23 (2021) 1078–1124. doi:10.1109/COMST.2021.3062546.

[2] A. Jøsang, R. Ismail, The Beta Reputation System, in: 15th Bled Electronic Commerce Conference, University of Maribor Press, Bled, Slovenia, 2002.

[3] A. Musaddiq, Y. B. Zikria, O. Hahm, H. Yu, A. K. Bashir, S. W. Kim, A Survey on Resource Management in IoT Operating Systems, IEEE Access 6 (2018) 8459–8482. doi:10.1109/ACCESS.2018.2808324.

[4] T. Grandison, M. Sloman, A survey of trust in internet applications, IEEE Communications Surveys Tutorials 3 (2000) 2–16. doi:10.1109/COMST.2000.5340804.

v				W	x				$\hat{W}$	$W - \hat{W}$	Cap
$[\mathcal{C}$	$\mathcal{T}$	$\mathcal{R}$	$\mathcal{S}]$	(B)	$[\mathcal{C}$	$\mathcal{T}$	$\mathcal{R}$	$\mathcal{S}]$	(B)	(B)	(%)
[1	1	1	1]	383	[2	4	0	3]	376	7	15
[1	1	1	1]	399	[2	4	0	4]	384	15	17
[1	1	1	1]	512	[2	5	0	5]	400	112	19
[1	1	1	1]	959	[5	10	0	9]	952	7	40
[1	1	1	1]	1 024	[5	10	0	10]	960	64	42
[1	1	1	1]	2 015	[9	18	1	18]	1 872	143	78
[1	1	1	1]	2 048	[9	18	2	18]	2 016	32	81
[1	1	1	1]	3 023	[9	18	8	18]	2 880	143	97
[1	1	1	1]	4 096	[9	18	9	18]	3 024	1 072	100
[9	1	1	1]	512	[3	0	0	4]	512	0	14
[1	9	1	1]	512	[2	5	0	5]	400	112	19
[1	1	9	1]	512	[2	3	1	3]	512	0	17
[1	1	1	9]	512	[2	5	0	5]	400	112	19
[9	9	1	1]	512	[3	4	0	0]	512	0	14

Table 4: Solutions to buffer sizes (x). Where input v represents the importance weighting for each region, input W represents the total amount of memory available, number of agents $|V| = 10$, and number of services $|S| = 2$. Output $\hat{W}$ is the amount of memory used.

[5] P. Taylor, L. Barakat, S. Miles, N. Griffiths, Reputation assessment: a review and unifying abstraction, The Knowledge Engineering Review 33 (2018) e6. doi:10.1017/S0269888918000097.

[6] W. L. Teacy, M. Luck, A. Rogers, N. R. Jennings, An efficient and versatile approach to trust and reputation using hierarchical bayesian modelling, Artificial Intelligence 193 (2012) 149–185. doi:10.1016/j.artint.2012.09.001.

[7] Y. Wang, I.-R. Chen, J.-H. Cho, J. J. P. Tsai, Trust-Based Task Assignment With Multiobjective Optimization in Service-Oriented Ad Hoc Networks, IEEE Transactions on Network and Service Management 14 (2017) 217–232. doi:10.1109/TNSM.2016.2636454.

[8] W. T. L. Teacy, J. Patel, N. R. Jennings, M. Luck, TRAVOS: Trust and Reputation in the Context of Inaccurate Information Sources, Autonomous Agents and Multi-Agent Systems 12 (2006) 183–198. doi:10.1007/s10458-006-5952-x.

[9] E. ElSalamouny, V. Sassone, An HMM-Based Reputation Model, in: A. I. Awad, A. E. Hassanien, K. Baba (Eds.), Advances in Security of Information and Communication Networks, Springer, 2013, pp. 111–121.

[10] Y. Wang, J. Vassileva, Bayesian network-based trust model, in: Proceedings IEEE/WIC International Conference on Web Intelligence, 2003, pp. 372–378. doi:10.1109/WI.2003.1241218.

[11] H. Fang, J. Zhang, M. Sensoy, N. M. Thalmann, A generalized stereotypical trust model, in: 2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications, 2012, pp. 698–705. doi:10.1109/TrustCom.2012.29.

[12] U. Jayasinghe, G. M. Lee, T. Um, Q. Shi, Machine Learning Based Trust Computational Model for IoT Services, IEEE Transactions on Sustainable Computing 4 (2019) 39–52.

[13] S. Pan, Z. Zhang, Z. Zhang, D. Zeng, Dependency-Aware Computation Offloading in Mobile Edge Computing: A Reinforcement Learning Approach, IEEE Access 7 (2019) 134742–134753. doi:10.1109/ACCESS.2019.2942052.

[14] M. Debe, K. Salah, M. H. U. Rehman, D. Svetinovic, IoT Public Fog Nodes Reputation System: A Decentralized Solution Using Ethereum Blockchain, IEEE Access 7 (2019) 178082–178093. doi:10.1109/ACCESS.2019.2958355.

[15] M. J. Probst, Sneha Kumar Kasera, Statistical trust establishment in wireless sensor networks, in: 2007 International Conference on Parallel and Distributed Systems, 2007, pp. 1–8.

[16] C. E. Player, N. Griffiths, Bootstrapping trust and stereotypes with tags, in: Proceedings of the 19th International Workshop on Trust in Agent Societies, Sao Paulo, Brazil, 2017.

[17] M. Bradbury, A. Jhumka, T. Watson, Trust Trackers for Computation Offloading in Edge-Based IoT Networks, in: IEEE INFOCOM, IEEE, Vancouver, BC, Canada, 2021, pp. 1–10. doi:10.1109/INFOCOM42981.2021.9488844.

[18] R. Hasan, M. Hossain, R. Khan, Aura: An incentive-driven ad-hoc IoT cloud framework for proximal mobile computation offloading, Future Generation Computer Systems 86 (2018) 821–835. doi:10.1016/j.future.2017.11.024.

[19] S. Podlipnig, L. Böszörmenyi, A Survey of Web Cache Replacement Strategies, ACM Computing Surveys 35 (2003) 374–398. doi:10.1145/954339.954341.

[20] J. Xu, Q. Hu, W.-C. Lee, D. L. Lee, Performance evaluation of an optimal cache replacement policy for wireless data dissemination, IEEE Transactions on Knowledge and Data Engineering 16 (2004) 125–139. doi:10.1109/TKDE.2004.1264827.

[21] E. J. O'Neil, P. E. O'Neil, G. Weikum, The LRU-K Page Replacement Algorithm for Database Disk Buffering, SIGMOD Rec. 22 (1993) 297–306. doi:10.1145/170036.170081.

[22] L. A. Bélády, A study of replacement algorithms for a virtual-storage computer, IBM Systems Journal 5 (1966) 78–101. doi:10.1147/sj.52.0078.

[23] I. Chen, J. Guo, F. Bao, Trust Management for SOA-Based IoT and Its Application to Service Composition, IEEE Transactions on Services Computing 9 (2016) 482–495. doi:10.1109/TSC.2014.2365797.

[24] L. Xiong, L. Liu, PeerTrust: supporting reputation-based trust for peer-to-peer electronic communities, IEEE Transactions on Knowledge and Data Engineering 16 (2004) 843–857. doi:10.1109/TKDE.2004.1318566.

[25] K. Xiao, Z. Gao, W. Shi, X. Qiu, Y. Yang, L. Rui, EdgeABC: An architecture for task offloading and resource allocation in the Internet of Things, Future Generation Computer Systems 107 (2020) 498–508. doi:10.1016/j.future.2020.02.026.

[26] C. Antonopoulos, A. Prayati, T. Stoyanova, C. Koulamas, G. Papadopoulos, Experimental evaluation of a WSN platform power consumption, in: 2009 IEEE International Symposium on Parallel & Distributed Processing, IPDPS '09, 2009, pp. 1–8. doi:10.1109/IPDPS.2009.5161185.

[27] Zolertia, Zolertia RE-Mote Revision B Internet of Things hardware development platform, for 2.4-GHz and 863-950MHz IEEE 802.15.4, 6LoWPAN and ZigBee® Applications, Datasheet ZOL-RM0x-B, Bar-

celona, Spain, 2016. V1.0.0.

[28] Nordic Semiconductor, nRF52840, Product Specification 4413_417, 2019. URL: https://infocenter.nordicsemi.com/pdf/nRF52840_PS_v1.1.pdf, V1.1.

[29] A. Avizzienis, J.-C. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE Transactions on Dependable and Secure Computing 1 (2004) 11–33. doi:10.1109/TDSC.2004.2.

[30] M. Bradbury, A. Jhumka, T. Watson, D. Flores, J. Burton, M. Butler, Threat-Modeling-Guided Trust-Based Task Offloading for Resource-Constrained Internet of Things, ACM Transactions on Sensor Networks 18 (2022) 41. doi:10.1145/3510424.

[31] M. R. Garey, D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman & Co., New York, NY, USA, 1979.

[32] A. Banks, E. Briggs, K. Borgendale, R. Gupta (Eds.), MQTT Version 5.0, OASIS Standard, Burlington, MA, USA, 2019. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.

[33] O. Eytan, D. Harnik, E. Ofer, R. Friedman, R. Kat, It's Time to Revisit LRU vs. FIFO, in: 12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20), USENIX Association, 2020.

[34] S. Martello, P. Toth, Knapsack Problems: Algorithms and Computer Implementations, John Wiley & Sons, Inc., USA, 1990.

[35] J. Höglund, S. Lindemer, M. Furuhed, S. Raza, PKI4IoT: Towards public key infrastructure for the Internet of Things, Computers & Security 89 (2020) 101658. doi:10.1016/j.cose.2019.101658.

[36] L. Sigrist, R. Ahmed, A. Gomez, L. Thiele, Harvesting-Aware Optimal Communication Scheme for Infrastructure-Less Sensing, ACM Trans. Internet Things 1 (2020). doi:10.1145/3395928.

[37] D. D. S. Braga, M. Niemann, B. Hellingrath, F. B. D. L. Neto, Survey on Computational Trust and Reputation Models, ACM Comput. Surv. 51 (2018). doi:10.1145/3236008.

[38] X. Gong, T. Yu, A. J. Lee, Bounding Trust in Reputation Systems with Incomplete Information, in: Proceedings of the Second ACM Conference on Data and Application Security and Privacy, CODASPY '12, Association for Computing Machinery, New York, NY, USA, 2012, pp. 125–132. doi:10.1145/2133601.2133617.

[39] M. Bradbury, A. Jhumka, T. Watson, Dataset for: Buffer Management for Trust Computation in Resource-constrained IoT Networks, 2020. doi:10.5281/zenodo.4353611.