Model 0

This paper presents the fundamental assumptions for the so called Model 0. Model 0 is the codename for the problem of anti Sybil attacks and potential defence against such attacks in the Golem's distributed marketplace, where reputation is local, there are no blockchain deposits, and it is easy to create new identity.

Introduction

Consider a P2P network with transactions like Golem Network. It is quite natural to assume that the providers are to be paid in arrears by the requestors for the work they have done. It is however a common problem in a P2P network of how to enforce such payments, or any payments for that matter. This is a double-edged sword as if we enforce payments in one way or another, how are we to protect the requestors against potential frauds, i.e., the providers delivering incorrect results.

In this paper, we assume a truly decentralized network. This implies that there are no atomic swaps, arbitrage, mediators, and no central or trusted services. In such a environment, in order to overcome the problem of enforcing payments, there are a few tools at our disposal; for example, we can use blockchains (or other consensus-based solutions), distributed reputation (or other distributed algorithms), TEEs, etc. In addition, we may even utilise external data sources making our solution more or less decentralized and trustless. In the model studied in this paper, we assume a raw P2P network, which essentially implies that any node can rely only on a local history of its own transactions/interactions with other nodes on the network.

Notice that a requestor has full control over the payments. Thus, it is the provider who is faced with a dilemma: will the requestor honour the unwritten contract between them and pay for the work done by the provider, or not. In this paper, we focus our attention on the problem of unknown requestors, i.e., requestors the provider has not yet collaborated with before. Dealing with unknown requestors is of particular importance, as it is a matter of delicate balance between letting the provider to expand its network of known requestors, and not getting paid by a fraudster requestor who assumes a one-time identity and does not pay for the work done. We refer to this problem as the risk of unknown requestors, and the goal for a provider is to minimise their losses without foregoing the exploration of unknown requestors. We emphasise the fact that we do not strive to protect a provider but to minimize their losses, which proves a hard problem in Model 0.

As a reference case, let us consider the following attack of a malicious party on the network. The attacker creates lots of identities and floods the network with computation tasks with no intention to pay. Allowing such a situation to linger for a prolonged period of time will surely destabilise and ultimately destroy the market. A desired reaction of honest nodes is that after being subjected to a noticable number of failures, the provider will then block all of the unknown requestors from further interaction, at least temporarily. However, in the process, the network will eventually get rigid and closed to newcomers, i.e., only nodes with reputation will be admitted to participate in the market. Unfortunately, it is natural that there will be a rotation of requestors resulting in the network slowly shrinking and the market collapsing in the long term.

In this paper, we propose an algorithm that assesses and manages the risk of unknown requestors relying only on local reputation.

Intuition and assumptions

We assume the following,

1. We are able to differentiate between known and unknown requestors. In particular, a known requestor is a requestor with history of transactions. Whether it is good or bad is less important than just the fact that it exists and is well known. Unknown requestors, on the other hand, have no history of transactions or are poorly known.
2. We consider unknown requestor a risk. This paper proposes a way to manage this risk.
3. In case of a sybil attack, it is likely there will be many unknown requestors. Here, we consider all unknown requestors as one requestor. This immediately implies that the ratio between the number of tasks accepted from known requestors and the number of tasks accepted from unknown requestors will not be influenced directly by the sheer number of task offers from unkown requestors. In other words, even if there are many such tasks, the influence on the ratio is negligible.
4. In general, poor reputation of a requestor can influence the provider's offer price or can influnce the provider's willingness to accept the task from this particular requestor. However, this is beyond the scope of this paper and will be handled separately in other work.
5. The greater the risk of unknown requestors, the smaller the number of tasks from unknown requestors that should be computed.
6. The computation of tasks from unknown requestors ends with either success or failure. (Success means that the task is paid. It is not obvious in Brass Golem Marketplace because of payment delay, but this paper does not cover this.)
7. The greater the number of failed tasks from unknown requestors, the greater the risk of unknown requestors.
8. The fewer the number of tasks from known requestors, the smaller the risk of unknown requestors.

The algorithm

The goal is to decide, as a provider, whether or not to send an offer for an advertised task. It is vital to note that this algorithm does not represent the whole decision process; it only focuses on unknown requestors. There are other decision structures, like constraints and price function, and this algorithm should be combined with them.

The node receives a task offer from the requestor. The requestor has the reputation vector R. This algoritm is not dependent on the structure of R. We consider a given function h which indicates how well the requestor is known based on its reputation R. $h(R) \in [0, 1]$. 0 means that the requestor is unknown, 1 means that the requestor is considered as known. The function h is not to be changed while the node is performing work.

Other constants and variables:

- S. Cumulative sum of successfully computed tasks that comes from unkown requestors. Tasks are measured in time of their execution. A task is successfully computed if it is paid. The sum S is subject to history smoothing.
- F. Cumulative sum of computed tasks with failure that comes from unkown requestors. Tasks are measured in time of their execution. A task is failed if the node started to compute it but the computation ends with an error or timeout or the computation is completed but is unpaid. The sum F is subject to history smoothing.
- Constant γ. Default value is 0.1, and it represents the history smoothing factor for S and F.
- t_0. The finish time of the last successful or failed task that comes from the unknown requestor. This is the time when α, β and τ are reset.
- τ. The sum of computed tasks that comes from known requestors since t_0. It does not matter if tasks are successful, failed, timed out, paid or unpaid. Tasks are measured in time of their execution.
- σ. The sum of computed tasks that comes from unknown requestors since t_0. It does not matter if tasks are successful, failed, timed out, paid or unpaid. Tasks are measured in time of their execution.
- α, β. The factors for the risk of unknown requestors.
- Constants κ, δ. Default values are $\kappa = 1$ and $\delta = 0.5$. δ represents the smallest frequency with which to accept tasks from unknown requestors, and κ is the velocity factor for lowering resistence towards unknown requestors with respect to time. These are the factors that represent risk aversion of the provider.
- Constant $\eta \geq 1$ (default value is $\eta = 1.5$) — an exponent shaping sensitivity of β to the failure rate. Our simulation shows that for $\eta > 1$ the algorithm is more efficient at limiting provider losses (at the expense of possibly rejecting more tasks from unknown requestors).
- Δt. It is the task computation time.

The initial values are.

- $S = 0$
- $F = 0$
- $t_0 = now$
- $\tau = 0$
- $\sigma = 0$
- $\alpha = 0$
- $\beta = \kappa(\frac{S+\delta}{S+F+\delta})^\eta$

We define difficulty for unknown requestors as an expression $e^{\alpha-\beta(now-t_0+\tau)}$. It is the probability that a task from an unknown requestor is rejected. Note the following.

- The difficulty decreases with respect to time and tasks from known requestors with speed β.
- The variables α, β, t_0, τ, σ are being updated from time to time. The difficulty as a function of time is not continuous.
- The difficulty can be greater than one. This means that temporarily tasks from unknown requestors are always rejected. After some time the difficulty will drop below one.
- After the variables are reset, a provider waits α/β time until difficulty drops below one.

The algorithm.

1. The node receives a task offer from a requestor. The requestor has reputation R.
2. If $random \leq h(R)$ then the requestor is considered known, otherwise it is considered unknown.
3. If the requestor is known then:

- the algorithm accepts the task,
- when task computation is done, regardless of the outcome, then $\tau := \tau + \Delta t$,
- the algorithm ends.

4. If $random < e^{\alpha-\beta(now-t_0+\tau)}$ then the task is rejected and the algorithm ends.
5. Otherwise, the task is accepted.
6. If the requestor assigns the node for computation and it starts computations then the node recalculates variables:

- $t_0 := now$
- $\tau := 0$
- $\alpha := \sigma$
- $\beta := \kappa(\frac{S+\delta}{S+F+\delta})^\eta$
- $\sigma := 0$

7. When the task is computed successfully or with failure, then the node recalculates variables:

- $\sigma := \sigma + \Delta t$
- $S := S \cdot e^{-\gamma \Delta t} + \frac{1-e^{-\gamma \Delta t}}{\gamma}[success?]$
- $F := F \cdot e^{-\gamma \Delta t} + \frac{1-e^{-\gamma \Delta t}}{\gamma}[failure?]$

Remarks

We solve the problem with partially known requestors by treating $h(R)$ as probability. This way the algorithm is much simplier and easier to maintain.

random is a random number from $[0, 1]$. At any time instant, the algorithm draws a new random number.

$[success?]$ and $[failure?]$ take values 0 or 1 depending on the last task outcome.

When algoritm states 'task is accepted', keep in mind that the task can be rejected outside of this algorithm due to some other constraints.

Notice that $e^{\alpha} > 1$. Thus, after the task is done for an unknown requestor, the provider waits α/β time until it can compute another task for an unknown requestor.

The work done for known requestors 'speeds up time' keeping the ratio of the work done for known and unknown requestors.

τ can be weighted. S and F can have different weights when calculating β.

When the network is flooded with fake tasks, the providers get failures almost every time. Then providers keep 'pinging' unknown requestors to check if the flood is over with frequency of $\kappa \frac{\delta}{\frac{1}{1-\gamma}+\delta}$ $(\approx 5\%)$.

There are two issues regarding updating the difficulty. There is a short time interval between sending an offer by a provider and selecting it by a requestor. The difficulty is updated upon successful selection. Meanwhile the provider can send more offers with very similar (low) difficulty to unknown requestors. It is a little issue for multi tasks providers. Subsequent offers should be considered with updated (high) difficulty by the provider. The second issue is with verification. Verification of the results takes time and the requestor has a given time limit for sending acceptance of the results. The variables S, F and σ are not updated upon finish of computations but upon receiving verification outcome. This is not a critical problem, just the variables are updated with a delay. And sybil attack with flood of unknown requestors and tasks is recognized with a delay also.

In case of Golem Brass, there still remains the problem of delayed payments.

In case of sybil attack with a flood of unknown requestors and tasks, a provider will pick up a task from unknown requestor soon after the difficulty drops below one and the provider is free. The above algorithm takes into account this situation.

The variables S and F refer to cumulative sums of execution time and fall under history forgetting. This is continuous history forgetting. This means that when a given task ends, then its execution time falls under history forgetting already then. Simply, if a given task just ended and its execution time is Δt then we calculate $\int_0^{\Delta t} e^{-\lambda t} dt$ for S or F respectively. This way the limit value for $S+F$ is $\frac{1}{\lambda}$. Putting it in other words. Assume that a provider have just completed a task and we have 'removed' all time intervals from the timeline when the provider was not busy. Then $S = \int_{-\infty}^{0} e^{-\lambda \cdot (now - t)} [success(t)?] dt$ where $[success(t)?] \in \{0, 1\}$ tests if the provider was computing a successfully completed task at the moment t. F respectively.