Brass Golem Marketplace

Authors: Marcin Benke, Łukasz Gleń

August 2018, Golem Factory

This paper presents Brass Golem Marketplace definition. There is no motivations, broader descriptions nor discussions just structure and assumptions.

Transaction framework

In this context a transaction is an agreement and execution of the agreement between Requestor and Provider on performing computation in exchange for a payment. Each transaction in Brass Golem marketplace is a one-to-one, the notion of a task is irrelevant, only a subtask that is subject of an agreement.

Model

1. Requestor sends public information on task to be computed. The message potentially reaches every Provider. The message includes task/subtasks estimation – timeout and min_perf. It also includes max_price for a subtask but it is questionable if it is relevant.
2. Provider sends private offer to Requestor. It includes offered price. Provider sends the message if and only if it is interested.
3. Requestor sends an agreement to selected Provider. It is a private message. It contains price for a subtask. Provider can reject the agreement in a short time frame for cancellation without consequences (like lower reputation).
4. Having an agreement Provider computes a subtask. Possible final statuses are: failed, timeout, rejected (due negative outcome of verification), accepted. Computation time is recorded.
5. Statuses for payments for accepted subtasks are: waiting, overdue, paid.

Protocol scheme

The basic, positive, path is as follows.

1. Broadcast. Requestor sends TaskHeader to Providers (it is not sent but for the sake of simplicity let us say that it is sent). TaskHeader contains `max_price`, `subtask_timeout`, `min_perf`.
2. WantToComputeTask. Provider responds with WantToComputeTask. The message contains Provider's price (an offer) for the subtask.
3. TaskToCompute. Requestor agrees on Provider's conditions and sends TaskToCompute. The message has notion of a formal agreement between Requestor and Provider, Requestor is obligated to pay for correct results and Provider is obligated to start computations immediately.

4. ReportComputedTask. Provider successfully finishes computation and sends the message ReportComputedTask.
5. Requestor accepts results.
6. Requestor pays.

Negative paths are as follows.

1. Provider is not interested or it does not satisfy requirements (like `min_perf`) - Provider does not respond to TaskHeader.
2. Task is already finished or all subtasks are already assigned - Requestor responds with CannotAssignTask to WantToComputeTask.
3. Requestor does not accept Provider's offer - Requestor does not respond to WantToComputeTask.
4. Provider cancels agreement - immediately after receiving TaskToCompute, Provider sends CannotComputeTask with a reason "offer cancelled". If the message is sent right after receiving TaskToCompute then the agreement is considered void and reputation should not change.
5. Subtask failed - Provider sends TaskFailed or CannotComputeTask (with reason other than "offer cancelled"); subtask computation is considered as failed.
6. Subtask timeout - Provider do not send ReportComputedTask on time or Requestor cannot download results.
7. Verification failed - Requestor verified Provider's results and the outcome is negative; Requestor rejects results.
8. No payment - Requestor do not pay for correct results.

Principles

1. Subtask is a subject of an agreement. In other words, Requestor pays for a subtask, not for used computational power. The consequence is that the value of payment is set before computations start.
2. The following Requestor's estimation holds: a node with `min_perf` performance completes the subtask in `subtask_timeout` time.

Conventions

1. `max_price` in TaskHeader, `price` in WantToComputeTask are per 1 hour of computation
2. `price` in TaskToCompute is calculated for whole subtask as below

$$\text{computation\_price} = \lfloor (\text{price} * \text{computation\_time} + 3599)/3600 \rfloor$$

Changes to Golem protocol

1. New reason in CannotComputeTask: "offer cancelled".
2. Implicite change in reaction: if CannotComputeTask("offer cancelled") is immediate then reputation is not decreased - agreement is assumed to never existed.

3. Implicite change: Requestor does not respond immediately on WantToComputeTask (Requestor pools offers).

Reputation on Requestor side

Our vision for the reputation handling on requestor side is discussed more extensively in the Local Reputation Proposal document, here we present a summary of the conclusions.

Measuring Efficiency

Upon successful subtask completion, the efficiency score should change according to relative computation velocity for that subtask:

$$v = \text{timeout} \ / \ \text{computation\_time}$$

intuitively corresponding to the number of times given provider would manage to complete the subtask in the time given.

The cumulative efficiency measure is then updated to weighted average of historical performance and recent computation velocity:

$$R_{t+1} = \psi R_t + (1 - \psi)v$$

where ψ is the history forgetting (aka smoothing) factor.

The default value for $\psi = 0.9$. The initial value for R is

$$R_0 = \min\{4, \text{provider.perf} \ / \ \text{requestor.min\_perf}\}$$

which corresponds to assuming that performance of an unknown Provider is its declared performance but capped by 4 x `min_perf`. The efficiency measure is calculated only for successfully computed subtasks, i.e. for subtasks receiving AckReportComputedTask, or possibly for subtasks with completed computations (not failures nor timeouts).

Efficacy

We track efficiency and efficacy separately. The value R as described in the previous section, is a measure of efficiency, while efficacy is tracked by means of a quality vector $Q = (s, t, f, r)$ consisting of rates of success (positive Requestor verification outcome), timeout, failure (other than timeout) and rejected (due negative Requestor verification outcome or no outcome).

This vector should be subject to history smoothing as well, so upon successful subtask completion the vector gets modfied like this:

$$Q_{t+1} = \psi Q_t + (1, 0, 0, 0)$$

The default value for ψ is 0.9.

Reputation on Provider side

From a provider's point of view, requestor reputation depends on how fast it's tasks can be computed, and how much of the work done gets paid. It consists of three factors:

R — relative efficiency defined exactly the same as on Requestor with initial value:

$$R_0 = \text{provider.perf / requestor.min\_perf}$$

V_{paid} – sum of prices (in GNT) paid for all Requestor's subtasks, in other words it is a sum of all Requestor's payments. This factor is not a subject to history forgetting.

V_{assigned} – sum of prices (in GNT; stated in TaskToCompute message) for subtasks assigned by this Requestor, excluding subtasks that Provider cancelled by sending CannotComputeTask message. This factor is not a subject to history forgetting.

Provider selection function

A requestor may choose providers for a given task basing on their reputation (hence basically effective performance) and price. The relative importance of these factors may vary from one requestor to another, we can however attempt to characterize it by price sensitivity factor $\alpha \in [0, 1]$ and compute provider score as

$$S_i = \frac{\alpha c_{max}}{c_i} + (1 - \alpha)R * q$$

where q is quality factor representing success ratio calculated from the quality vector (reputation) as follows.

$$q = \frac{1 + s}{d + s + t + f + r} \cdot \frac{1}{q^*}$$

where $Q = (s, t, f, r)$ and $d \geq 1$ is the distrust factor towards unknown providers and

$$q^* = \frac{1 + \frac{1}{1-\psi}}{d + \frac{1}{1-\psi}}$$

is maximal value of q, therefore it normalizes q. The default value for d is 5.

The default Provider selection function is greedy policy – Providers with the highest scores are chosen.

We implement also an optional selection function (Golem client can switch to this function via command line parameters). Consider set of offers J with scores $\{S_j \mid j \in J\}$. The probablity p_j of chosing provider j is (note that the score is always a positive number)

$$p_j = \frac{e^{\frac{-\lambda}{S_j}}}{\sum_{j \in J} e^{\frac{-\lambda}{S_j}}}$$

where λ is a greed factor (with λ equals $+\infty$ it converges to the greedy policy).

Requestor's contraints

The Requestor ignores any offer from a given Provider if the Provider's reputation (quality factor) q defined above falls below the limit set by the user.

Provider price function

Base price calculation

Provider is interested in maximizing its effective price per hour. Hence the main variable is expected revenue per hour r (what I expect to get **paid** for an hour of computation). Obviously, this varies with varying demand and supply of computational power. We propose the following strategy for price adjustment: let m be minimal price per hour (think operating costs), t_1 be the time of last subtask completion (if any, otherwise the provider start time) and let r_1 be the effective price calculated at that time. Let t_2 be the current time (when we consider submitting an offer). We propose that the expected profit f fall exponentially with provider "thirst" (time spent unemployed), i.e.

$$\text{thirst} = t_2 - t_1$$
$$f_2 = e^{-\alpha(\text{thirst})} f_1$$

than the revenue is

$$r_2 = m(1 + f_2)$$

On the other hand, sucessful employment (reported subtask) reduces provider thirst and lets it update its profit expectations upward:

$$f_2 = e^{\beta(t_2 - t_s)} f_s$$

where t_s is the time when this subtask computation started and t_2 is current time – it is time when subtask computation ends.

Intuitively, the $\frac{\beta}{\alpha}$ ratio corresponds to the expected employment ratio and their actual values determine how fast the prices rise or fall with time.

Proposed default values are $\alpha = 10^{-5}s^{-1}$ and $\beta = 4\alpha$ (s^{-1} means time is measured in seconds). Initial value of f is 0.2.

Subtask price function

As observed earlier, a provider is interested in maximizing its effective price per hour. Hence from its point of view, requestor reputation score should reflect the ratio of effective and transaction price. Conversely to achieve effective revenue r, it sets the transaction price p according to requestor score S:

$$p = \frac{r}{S}$$

The score itself depends on the following factors:

- R — relative efficiency (ratio of nominal to effective computation time) described in the section on reputation,
- Q — ratio of subtasks paid to subtasks assigned (in terms of the GNT value, taking into account some credit c for payments in progress)
- V_{paid} — total GNT value of subtasks paid by this Requestor (see above)
- V_{assigned} — total GNT value of subtasks assigned by this Requestor (see above)

$$R_{t+1} = \psi * R_t + (1 - \psi)\frac{\text{timeout}}{\text{time}}$$

$$Q = min\left(\frac{m + V_{\text{paid}} + c}{m + V_{\text{assigned}}}, 1\right)$$

$$S = QR$$

The default value for c is m — one hour of work with minimal price. The c stands for credit — amount of (temporarily) unpaid work Provider is willing to tolerate.

If the calculated price exceeds task `max_price`, the latter is used.

Provider's constraints

1. Provider does not send any offer to a given Requestor if there are more than k hours of work to be paid — more precisely, it is sum of computation time of all assigned subtasks that are not failed, timed out, rejected, paid or overdue in payment. Default value for k is 3.
2. Provider does not send any offer to a given Requestor if the Requestor's reputation Q defined above falls below the limit set by the user.