Contract-Based Load Management in Federated Distributed Systems[*]

Magdalena Balazinska, Hari Balakrishnan, and Mike Stonebraker
MIT Computer Science and Artificial Intelligence Lab
`http://nms.lcs.mit.edu/projects/medusa/`

Abstract

This paper focuses on load management in loosely-coupled federated distributed systems. We present a distributed mechanism for moving load between autonomous participants using bilateral contracts that are negotiated offline and that set bounded prices for moving load. We show that our mechanism has good incentive properties, efficiently redistributes excess load, and has a low overhead in practice.

Our load management mechanism is especially well-suited for distributed stream-processing applications, an emerging class of data-intensive applications that employ a "continuous query processing" model. In this model, streams of data are processed and composed continuously as they arrive rather than after they are indexed and stored. We have implemented the mechanism in the *Medusa* distributed stream processing system, and we demonstrate its properties using simulations and experiments.

1 Introduction

Many distributed systems are composed of loosely coupled autonomous nodes spread across different administrative domains. Examples of such federated systems include Web services, cross-company workflows where the end-to-end services require processing by different organizations [3, 21], and peer-to-peer systems [8, 23, 30, 45]. Other examples are computational grids composed of computers situated in different domains [4, 16, 44], overlay-based computing platforms such as Planetlab [35], and data-intensive stream processing systems [1, 2, 5, 6, 7] that can be distributed across different domains to provide data management services for data streams.

Federated operation offers organizations the opportunity to pool their resources together for common benefit. Participants can compose the services they provide into more complete end-to-end services. Organizations can also cope with load spikes without individually having to maintain and administer the computing, network, and storage resources required for peak operation.

[*]This material is based upon work supported by the National Science Foundation under Grant No. 0205445. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Autonomous participants, however, do not collaborate for the benefit of the whole system, but rather aim to maximize their own benefit. A natural way to architect a federated system is thus as a *computational economy*, where participants provide resources and perform computing for each other in exchange for payment.[1]

When autonomous participants are also real economic entities, additional constraints come into play. The popularity of bilateral agreements between Internet Service Providers (ISPs) demonstrates that participants value and even require privacy in their interactions with each other. They also practice price and service discrimination [24], where they offer different qualities of service and different prices to different partners. For this purpose, ISPs establish bilateral Service Level Agreements, where they define confidential details of the *custom* SLA and prices that one partner offers another.

In this paper, we present a distributed mechanism for managing load in a federated system. Our mechanism is inspired on the manner in which ISPs collaborate. Unlike other computational economies that implement global markets to set resource prices at runtime, our mechanism is based on *private pairwise contracts* negotiated offline between participants. Contracts set tightly *bounded prices* for migrating each unit of load between two participants and specify the set of tasks that each is willing to execute on behalf of the other. We envision that contracts will be extended to contain additional clauses further customizing the offered services (e.g., performance and availability guarantees). In contrast to previous proposals, our mechanism (1) provides privacy to all participants regarding the details of their interactions with others, (2) facilitates service customization and price discrimination, (3) provides a simple and lightweight runtime load management using price pre-negotiation, and (4) has good system-wide load balance properties.

With this *bounded-price mechanism*, runtime load transfers occur only between participants that have pre-negotiated contracts, and at a unit price within the contracted range. The load transfer mechanism is simple: a participant moves load to another if the local processing cost is larger than the payment it would have to make to

[1]Non-payment models, such as bartering, are possible too. See Section 2 for details.

another participant for processing the same load (plus the migration cost).

Our work is applicable to a variety of federated systems, and is especially motivated by *distributed stream processing applications*. In these applications, data streams are continuously pushed to servers, where they undergo significant amounts of processing including filtering, aggregation, and correlation. Examples of applications where this "push" model for data processing is appropriate include financial services (*e.g.*, price feeds), medical applications (*e.g.*, sensors attached to patients), infrastructure monitoring (*e.g.*, computer networks, car traffic), and military applications (*e.g.*, target detection).

Stream processing applications are well-suited to the computational economy provided by a federated system. Data sources are often distributed and belong to different organizations. Data streams can be composed in different ways to create various services. Stream processing applications also operate on large volumes of data, with rates varying with time and often exceeding tens of thousands of messages per second. Supporting these applications thus requires dynamic load management. Finally, because the bulk of the processing required by applications can be expressed with standard well-defined operators, load movements between autonomous participants does not require full-blown process migration.

We have designed and implemented the bounded-price mechanism in *Medusa*, a federated distributed stream-processing system. Using analysis and simulations, we show that the mechanism provides enough incentives for selfish participants to handle each other's excess load, improving the system's load distribution. We also show that the mechanism efficiently distributes excess load when the aggregate load both underloads and overloads total system capacity and that it reacts well to sudden shifts in load. We show that it is sufficient for contracts to specify a small price-range in order for the mechanism to produce acceptable allocations where (1) either *no* participant operates above its capacity, or (2) if the system as a whole is overloaded, then *all* participants operate above their capacity. We further show that the mechanism works well even when participants establish heterogeneous contracts at different unit prices with each other.

We discuss related work in the next section. Section 3 presents the bounded-price load management mechanism and Section 4 describes its implementation in Medusa. We present several simulation and experimental results in Section 5 and conclude in Section 6.

2 Related Work

Cooperative load sharing in distributed systems has been widely studied (see, *e.g.*, [10, 19, 22, 25, 41]). Approaches most similar to ours produce optimal or near-optimal allocations using *gradient-descent*, where nodes exchange load among themselves producing successively less costly allocations. In contrast to these approaches, we focus on environments where participants are directed by self-interest and not by the desire to produce a system-wide optimal allocation.

As recent applications frequently involve independently administered entities, more efforts have started to consider participant selfishness. In mechanism design (MD) [20, 33], agents reveal their costs to a central entity that computes the optimal allocation and a vector of compensating payments. Agents seek to maximize their utility computed as the difference between payment received and processing costs incurred. Allocation and payment algorithms are designed to optimize agents utility when the latter reveal their true costs.

In contrast to pure mechanism design, algorithmic mechanism design (AMD) [29, 32] additionally considers the computational complexity of mechanism implementations. Distributed algorithmic mechanism design (DAMD) [12, 14] focuses on distributed implementations of mechanisms, since in practice a central optimizer may not be implementable. Previous work on DAMD schemes includes BGP-based routing [12] and cost-sharing of multicast trees [13]. These schemes assume that participants correctly execute payment computations. In contrast, our load management mechanism is an example of a DAMD scheme that does not make any such assumption because it is based on bilateral contracts.

Researchers have also proposed the use of economic principles and market models for developing complex distributed systems [27]. Computational economies have been developed in application areas such as distributed databases [43], concurrent applications [46], and grid computing [4, 16, 44]. Most approaches use pricing [4, 9, 15, 16, 39, 43, 46]: resource consumers have different price to performance preferences and are allocated a budget. Resource providers hold auctions to determine the price and allocation of their resources. Alternatively, resource providers bid for tasks [39, 43], or adjust their prices iteratively until demand matches supply [15].

These approaches to computational economies require participants to hold and participate in auctions for every load movement, thus inducing a large overhead. Variable load may also make prices vary greatly and lead to frequent re-allocations [15]. If the cost of processing clusters of tasks is different from the cumulative cost of independent tasks, auctions become combinatorial [31, 34][2], complicating the allocation problem. If auctions are held by overloaded agents, underloaded agents have the choice to participate in one or many auctions simultaneously, leading to complex market clearance and exchange mechanisms [29]. We avoid these complexities by bounding

[2]In a combinatorial auction, multiple items are sold concurrently. For each bidder, each subset of these items represents a different value.

the variability of runtime resource prices and serializing communications between partners. In contrast to our approach, computational economies also make it significantly more difficult for participants to offer different prices and different service levels to different partners.

As an alternative to pricing, recent approaches propose to base computational economies on *bartering*. SHARP [17] is an infrastructure that enables peers to securely exchange tickets that provide access to resources. SHARP does not address the policies that define how the resources should be exchanged. Chun *et al.* [8] propose a computational economy based on SHARP. In their system, peers discover required resources at runtime and trade resource tickets. A ticket is a soft claim on resources and can be rejected resulting in zero value for the holder. In contrast, our pairwise agreements do not specify any resource amounts and peers pay each other only for the resources they actually use.

Service level agreements (SLAs) are widely used for Web services and electronic commerce [3, 21, 37]. The contract model we propose fits well with these SLA infrastructures.

In P2P systems, peers offer their resources to each other for free. Schemes to promote collaboration use reputation [23], accounting [45], auditing [30], or strategyproof computing [28] to eliminate "free-riders" who use resources without offering any in return. In contrast, we develop a mechanism for participants that require tight control over their collaborations and do not offer their resources for free.

3 The Bounded-Price Mechanism

In this section, we define the load management problem in federated distributed systems, present the bounded-price mechanism and discuss its properties.

3.1 Problem Statement

We are given a system comprised of a set S of autonomous *participants* each with computing, network, and storage resources, and a time varying set K of heterogeneous *tasks* that impose a load on participants' resources. Each task is considered to originate at a participant where it is submitted by a client. Since we only examine interactions between participants, we use the terms *participant* and *node* interchangeably. Tasks can be aggregated into larger tasks or split into subtasks. If the load imposed by a task increases, the increase can thus be treated as the arrival of a new task. Similarly, a load decrease can be considered as the termination of a task. We discuss tasks further in Section 4.

For each participant, the load imposed on its resources represents a cost. We define a real-valued *cost function* of each participant i as:

$$\forall i \in S, \forall \mathrm{taskset_i} \subseteq K,\ \ D_i : \mathrm{taskset_i} \rightarrow \mathcal{R} \quad (1)$$

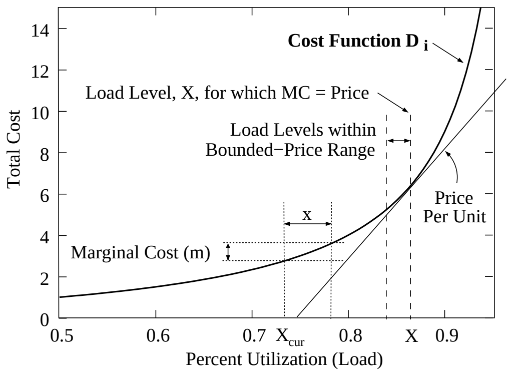


Figure 1: Prices and processing costs.

where $\mathrm{taskset}_i$ is the subset of tasks in K running at i. This cost depends on the load imposed by the tasks. Each participant monitors its own load and computes its processing cost. There are an unlimited number of possible cost functions and each participant may have a different one. We assume, however, that this cost is a *monotonic* and *convex* function. Indeed, for many applications that process messages (e.g., streams of tuples), an important cost metric is the per message processing delay. For most scheduling disciplines this cost is an increasing and convex function, reflecting the increased difficulty in offering low delay service at higher load. Figure 1 illustrates such cost function for a single resource. We revisit Figure 1 further throughout the section.

Participants are selfish and aim to maximize their utility, computed as the difference between the processing cost, $D_i(\mathrm{taskset_i})$, and the payment they receive for that processing. When a task originates at a participant, it has a constant per-unit load value to that participant (this value could be, for instance, the price paid by the participant's clients for the processing). When a task comes from another participant, the payment made by that participant defines the task's value.

Each participant has a maximum load level corresponding to a maximum cost, above which the participant considers itself overloaded. The goal of a load management mechanism is to ensure that no participant is overloaded, when spare capacity exists. If the whole system is overloaded, the goal is to use as much of the available capacity as possible. We seek a mechanism that produces an *acceptable allocation*:

Definition: An **acceptable allocation** is a task distribution where (1) *no* participant is above its capacity threshold, *or* (2) *all* participants are at or above their capacity thresholds if the total offered load exceeds the sum of the capacity thresholds.

Because the set of tasks changes with time, the allocation problem is an online optimization. Since the system is a federation of loosely coupled participants, no single entity can play the role of a central optimizer and the implementation of the mechanism must be distributed. We further examine the mechanism design aspects of our approach in Section 3.3.

In our scheme, load movements are based on *marginal costs*, $\mathrm{MC}_i : (u, \mathrm{taskset_i}) \to \mathcal{R}$ defined as the incremental cost for node i of running task u given its current $\mathrm{taskset_i}$. Figure 1 shows the marginal cost m caused by adding load x, when the current load is X_{cur}. Assuming the set of tasks in $\mathrm{taskset_i}$ imposes a total load X_{cur} and u imposes load x, then $\mathrm{MC}(u, \mathrm{taskset_i}) = m$. If x is one unit of load, we call m the *unit marginal cost*.

3.2 Model and Algorithms

We propose a mechanism to achieve acceptable allocations based on bilateral contracts: participants establish contracts with each other by negotiating offline a set of tightly bounded prices for each unit of load they will move in each direction.

Definition: A **contract** $\mathcal{C}_{i,j}$ between participants i and j defines a price range: $[\mathrm{min\_price}(\mathcal{C}_{i,j}), \mathrm{max\_price}(\mathcal{C}_{i,j})]$, that constrains the runtime price paid by participant i for each unit of load given to j.

Participants must mutually agree on what one unit of processing, bandwidth, and storage represent. Different pairs of participants may have contracts specifying different unit prices. There is at most one contract for each pair of participants and each direction. Participants may periodically renegotiate, establish, or terminate contracts offline. We assume that the set of nodes and contracts form a connected graph. The set of a participant's contracts is called its $\mathrm{contractset}$. We use C to denote the maximum number of contracts that any participant has.

At runtime, participants that have a contract with each other may perform *load transfers*. Based on their load levels, they agree on a definite unit price, $\mathrm{price}(\mathcal{C}_{i,j})$, within the contracted price-range, and on a set of tasks, the $\mathrm{moveset}$, that will be transferred. The participant offering load also pays its partner a sum of $\mathrm{price}(\mathcal{C}_{i,j}) * \mathrm{load}(\mathrm{moveset})$.

3.2.1 Fixed-Price Contracts

We first present the *fixed-price mechanism*, where $\mathrm{min\_price}(\mathcal{C}_{i,j}) = \mathrm{max\_price}(\mathcal{C}_{i,j}) = \mathrm{FixedPrice}(\mathcal{C}_{i,j})$. With fixed-price contracts, if the marginal cost per unit of load of a task is higher than the price in a contract, then processing that task locally is more expensive than paying the partner for the processing. Conversely, when a task's marginal cost per unit of load is below the price specified in a contract, then accepting that task results in a greater payment than cost increase.

Given a set of contracts, we propose a load management protocol, where each participant concurrently runs

```
00. PROCEDURE OFFER_LOAD:
01. repeat forever:
02.     sort(contractset on price(contractset_j) ascending)
03.     foreach contract $C_j \in$ contractset:
04.         offerset $\leftarrow \emptyset$
05.         foreach task $u \in$ taskset
06.             total_load $\leftarrow$ taskset $-$ offerset $- \{u\}$
07.             if $\mathrm{MC}(u, \mathrm{total\_load}) > \mathrm{load}(u) * \mathrm{price}(\mathcal{C}_j)$
08.                 offerset $\leftarrow$ offerset $\cup \{u\}$
09.         if offerset $\neq \emptyset$
10.             offer $\leftarrow (\mathrm{price}(\mathcal{C}_j), \mathrm{offerset})$
11.             (resp, acceptset) $\leftarrow$ send_offer($j$, offer)
12.             if resp = accept and acceptset $\neq \emptyset$
13.                 transfer($j$, $\mathrm{price}(\mathcal{C}_j)$, acceptset)
14.                 break foreach contract
15.     wait $\Omega$ time units
```

Figure 2: Algorithm for shedding excess load.

one algorithm for shedding excess load (Figure 2) and one for taking on new load (Figure 3).

The basic idea in shedding excess load is for an overloaded participant to select a maximal set of tasks from its $\mathrm{taskset_i}$ that cost more to process locally than they would cost if processed by one of its partners and offer them to that partner. Participants can use various algorithms and policies for selecting these tasks. We present a general algorithm in Figure 2. If the partner accepts even a subset of the offered tasks, the accepted tasks are transferred. An overloaded participant could consider its contracts in any order. One approach is to exercise the lower-priced contracts first with the hope of paying less and moving more tasks. In this paper, we ignore the task migration costs. These costs should, however, be considered before an offer is sent by imposing a minimum threshold between the difference in the local and remote processing costs.

Procedure $\mathrm{OFFER\_LOAD}$ waits between load transfers to let local load level estimations (e.g., exponentially weighted moving averages) catch-up with the new average load level. If no transfer is possible, a participant retries to shed load periodically. Alternatively, the participant may ask its partners to notify it when their loads decrease sufficiently to accept new tasks.

In procedure $\mathrm{ACCEPT\_LOAD}$ (Figure 3), each participant continuously accumulates load offers and periodically accepts subsets of offered tasks, examining the higher unit-price offers first. Since accepting an offer results in a load movement (because offers are sent to one partner at the time), the participant keeps track of all accepted tasks in the $\mathrm{potentialset}$ and responds to both accepted and rejected offers. Participants that accept a load offer cannot cancel transfers and move tasks back when load conditions change. They can, however, use their own contracts to move load further or to move it back.

There are several advantages in serializing communication between participants rather than having them offer their load simultaneously to all their partners. The

```
00. PROCEDURE ACCEPT_LOAD:
01. repeat forever:
02.     offers $\leftarrow \emptyset$
03.     for $\Omega$ time units or while (movement = true)
04.         for each new offer received, new_offer:
05.             offers $\leftarrow$ offers $\cup$ {new_offer}
06.     sort(offers on price(offers$_i$) descending)
07.     potentialset $\leftarrow \emptyset$
08.     foreach offer $o_i \in$ offers
09.         acceptset $\leftarrow \emptyset$
10.         foreach task $u \in$ offerset($o_i$)
11.             total_load $\leftarrow$ taskset $\cup$ potentialset $\cup$ acceptset
12.             if MC($u$, total_load) $<$ load($u$) $*$ price($o_i$)
13.                 acceptset $\leftarrow$ acceptset $\cup$ {$u$}
14.         if acceptset $\neq \emptyset$
15.             potentialset $\leftarrow$ potentialset $\cup$ acceptset
16.             resp $\leftarrow$ (accept, acceptset)
17.             movement $\leftarrow$ true
18.         else resp $\leftarrow$ (reject, $\emptyset$)
19.         respond($o_i$, resp)
```

Figure 3: Algorithm for taking additional load.

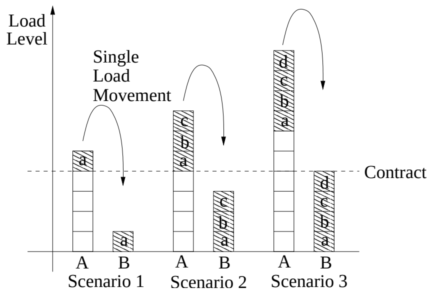


Figure 4: Three load movement scenarios for two partners.

communication overhead is lower but most importantly, the approach prevents possible overbooking as partners always receive the load they accept. This in turn allows participants to accept many offers at once. The only drawback is a slightly longer worst-case convergence time.

Figure 4 illustrates three load movement scenarios. If participant A has one or more tasks for which its marginal cost per unit of load exceeds the price in its contract with B, these tasks are moved in a single transfer (scenarios 1 and 2). Only those tasks, however, for which the marginal cost per unit of load at B does not exceed the price in the contract are transferred (scenario 3).

3.2.2 Setting Up Fixed-Price Contracts

With an approach based on fixed prices, the only tunable parameters are the unit prices set in contracts. When a participant negotiates a contract to shed load, it must first determine its maximum desired load level X, and the corresponding marginal cost per unit of load. This marginal cost is also the maximum unit price that the participant should accept for a contract. For any higher price, the participant risks being overloaded and yet unable to shed load. Any price below that maximum is acceptable. Figure 1 illustrates, for a single resource and a strictly convex function, how a load level X maps to a unit price. In general, this price is the gradient of the cost function evaluated at X.

When a participant negotiates a contract to accept load, the same maximum price rule applies since the participant will never be able to accept load once it is overloaded itself. A participant, however, should not accept a price that is too low because such price would prevent the participant from accepting new load even though it might still have spare capacity. The participant should rather estimate its expected load level, select a desired load level between the expected and maximum levels and negotiate the corresponding contract price.

Participants may be unwilling to move certain tasks to some partners due to the sensitive nature of the data processed or because the tasks themselves are valuable intellectual property. For this purpose, contracts can also specify the set of tasks (or types of tasks) that may be moved, constraining the runtime task selection. In offline agreements, participants may also prevent their partners from moving their operators further thus constraining the partner's task selections.

To ensure that a partner taking over a task provides enough resources for it, contracts may also specify a minimum per-message processing delay (or other performance metric). A partner must meet these constraints or pay a monetary penalty. Such constraints are commonplace in SLAs used for Web services, inter-company workflows or streaming services [18]. Infrastructures exist to enforce them through automatic verification [3, 21, 37, 38]. In the rest of this paper, we assume that such infrastructure exists and that monetary penalties are high enough to discourage any contract breaches. To avoid breaching contracts when load increases, participants may prioritize tasks already running over newly arriving tasks.

3.2.3 Extending the Price Range

Fixed-price contracts do not always produce acceptable allocations. For instance, load cannot propagate through a chain of identical contracts. A lightly loaded node in the middle of a chain accepts new tasks as long as its marginal cost is *strictly below* the contract price. The node eventually reaches maximum capacity (as defined by the contract prices) and refuses additional load. It does not offer load to partners that might have spare capacity, though, because its unit marginal cost is still lower than any of its contract prices. Hence, if all contracts are identical, a task can only propagate one hop away from its origin.

To achieve acceptable allocations for all configurations, participants thus need to specify a *small range of prices*, $[\text{FixedPrice} - \Delta; \text{FixedPrice}]$, in their contracts. Such price range allows a participant to forward load from an overloaded partner to a more lightly loaded one by accepting tasks at a higher price and offering them at a lower price. When a contract specifies a small price-range, for each load transfer, partners must dynamically negotiate the final unit price within the range. Since a fixed unit price equals the gradient (or derivative) of the cost curve at some load level, a price range converts into a load level interval as illustrated in Figure 1. The price range is the difference in the gradients of the cost curve at interval boundaries.

We now derive the minimal contract price-range that ensures convergence to acceptable allocations. We analyze a network of homogeneous nodes with identical contracts. We explore heterogeneous contracts through simulations in Section 5. For clarity of exposition, we also assume in the analysis that all tasks are identical to the smallest migratable task, u and *impose the same load*. We use ku to denote a set of k tasks.

We define δ_k as the decrease in unit marginal cost due to removing k tasks from a node's $\mathrm{taskset}$:

$$\delta_k(\mathrm{taskset}) = \frac{\mathrm{MC}(u, \mathrm{taskset} - u) - \mathrm{MC}(u, \mathrm{taskset} - (k+1)u)}{\mathrm{load}(u)} \quad (2)$$

δ_k is thus approximately the difference in the cost function gradient evaluated at the load level including and excluding the k tasks.

Given a contract with price, $\mathrm{FixedPrice}$, we define $\mathrm{taskset^F}$ as the maximum set of tasks that a node can handle before its per-unit-load marginal cost exceeds $\mathrm{FixedPrice}$ and triggers a load movement. I.e., $\mathrm{taskset^F}$ satisfies: $\mathrm{MC}(u, \mathrm{taskset^F} - u) \le \mathrm{load(u)} * \mathrm{FixedPrice}$ and $\mathrm{MC}(u, \mathrm{taskset^F}) > \mathrm{load(u)} * \mathrm{FixedPrice}$.

With all contracts in the system specifying the same price range, $[\mathrm{FixedPrice} - \Delta, \mathrm{FixedPrice}]$ such that $\Delta = \delta_1(\mathrm{taskset^F})$, any task can now travel two hops. A lightly loaded node accepts tasks at $\mathrm{FixedPrice}$ until its load reaches that of $\mathrm{taskset^F}$. The node then alternates between offering one task u at price $\mathrm{FixedPrice} - \delta_1(\mathrm{taskset^F})$ and accepting one task at $\mathrm{FixedPrice}$. This scenario is shown in Figure 5. Similarly, for load to travel through a chain of $M + 1$ nodes (or M transfers) the price range must be at least $\delta_{M-1}(\mathrm{taskset^F})$. The jth node in such a chain alternates between accepting a task at price $\mathrm{FixedPrice} - \delta_{j-1}(\mathrm{taskset^F})$ and offering it at price $\mathrm{FixedPrice} - \delta_j(\mathrm{taskset^F})$.

A larger price range speeds-up load movements through a chain because more tasks can be moved at each step. With a larger price range, however, nodes unit marginal costs are more likely to fall within the dynamic range requiring a price negotiation. A larger range thus increases runtime overhead, price volatility and the number of re-allocations caused by small load variations. Our goal is therefore to keep the range as small as possible and extend it only enough to ensure convergence to acceptable allocations.

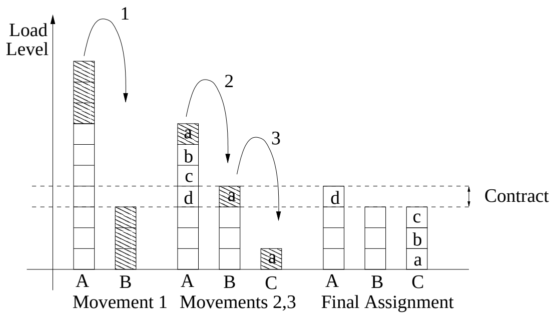


Figure 5: Load movements between three nodes using a small price-range.

For underloaded systems to always converge to acceptable allocations, tasks must be able to travel as far as the diameter M of the network of contracts. The minimal price range should then be $\delta_{M-1}(\mathrm{taskset^F})$.

Lemma 1 *In a network of homogeneous nodes, tasks, and contracts, to ensure convergence to acceptable allocations in underloaded systems, the unit price range in contracts must be at least:* $[\mathrm{FixedPrice} - \delta_{M-1}(\mathrm{taskset^F}), \mathrm{FixedPrice}]$*, where* M *is the diameter of the network of contracts and* $\mathrm{taskset^F}$ *is the set of tasks that satisfies* $\mathrm{MC}(u, \mathrm{taskset^F} - u) \le \mathrm{load(u)} * \mathrm{FixedPrice}$ *and* $\mathrm{MC}(u, \mathrm{taskset^F}) > \mathrm{load(u)} * \mathrm{FixedPrice}$*.*

When the system is overloaded, a price range does not lead to an acceptable-allocation (where $\forall i \in S : D_i(\mathrm{taskset_i}) \ge D_i(\mathrm{taskset_i^F})$). In the final allocation, some participants may have a marginal cost as low as $\mathrm{FixedPrice} - \delta_M(\mathrm{taskset_i^F})$ (wider ranges do not improve this bound). For overloaded systems, price-range contracts therefore achieve *nearly acceptable allocations* defined as:

Definition: A **nearly acceptable allocation** satisfies $\forall i \in S : D_i(\mathrm{taskset_i}) > D_i(\mathrm{taskset_i^F} - Mu)$

Price ranges modify the load management protocol as follows. Initial load offers are made at the lowest price. If no task can be accepted, the partner proposes a higher price. Upon receiving such a counter-offer, if the new price is still its best alternative, a participant recomputes the $\mathrm{offerset}$. If the set is empty, it suggests a new price in turn. Negotiation continues until both parties agree on a price or no movement is possible. Other negotiation schemes are possible.

3.3 Properties

The goal of mechanism design [33] is to implement an optimal system-wide solution to a decentralized optimization problem, where each agent holds an input parameter to the problem and prefers certain solutions over others. In our case, agents are participants and optimization parameters are participants' cost functions and original sets of tasks. The system-wide goal is to achieve an acceptable allocation while each participant tries to optimize its utility. Our mechanism is *indirect*: participants reveal their costs and tasks indirectly by offering and accepting tasks rather than announcing their costs directly to a central optimizer or to other agents.

A mechanism defines the set of strategies S available to agents and an outcome rule, $g : S^N \rightarrow O$, that maps all possible combinations of strategies adopted by agents to an outcome O. With fixed-price contracts, the runtime strategy-space of participants is reduced to only three possible strategies: (1) accept load at the pre-negotiated price, (2) offer load at the pre-negotiated price, or (3) do neither.[3] The desired outcome is an acceptable allocation.

Similarly to the definition used in Sandholm *et. al.* [40], our mechanism is *individual rational* (i.e., a participant may not decrease its utility by participating) on a per load movement basis. Each agent increases its utility by accepting load when the price exceeds the per-unit-load marginal cost (because in that case the increase in payment, p_i, exceeds the increase in cost, D_i) and offer load in the opposite situation. For two agents and one contract this strategy is also *dominant* because compared to any other strategy, it optimizes an agent's utility independently of what the other agent does.

For multiple participants and contracts, the strategy space is richer. Participants may try to optimize their utility by accepting too much load with the hope of passing it on at a lower price. Assuming, however, that participants are highly *risk-averse*: they are unwilling to take on load unless they can process it at a lower cost themselves because they risk paying monetary penalties, the strategy of offering and accepting load *only* when marginal costs are strictly higher or lower than prices respectively, is an optimal strategy. This strategy is not dominant, though, because it is technically possible that a participant has a partner that always accepts load at a low price. In specific situations, the order in which participants contact each other may also change their utility due to simultaneous moves by other participants.

These properties also hold for price-range contracts, when participants' marginal costs are far from range boundaries. Within a range, participants negotiate, thus revealing their costs more directly. Reaching an agreement is individual-rational since moving load at a price between participants' marginal costs increases both their utilities. Partners thus agree on a price within the range, when possible.

Our mechanism is also a distributed algorithmic mechanism (DAM) [12, 14] since the implementation is (1) distributed: there is no central optimizer, and (2) algorithmic: the computation and communication complexities are both polynomial time, as we show below. Because our mechanism is indirect, it differs from previous DAMD approaches [12, 14] that focus on implementing the same payment computation as would a central optimizer but in an algorithmic and distributed fashion. An important assumption made in these implementations is that agents are either separate from the entities computing the payment functions [14] or that they compute the payments honestly [12]. Our mechanism does not need to make any such assumption.

Because each load transfer takes place only if the marginal cost of the node offering load is strictly greater than that of the node accepting the load, successive allocations strictly decrease the sum of all costs. Under constant load, movements thus always eventually stop. If all participants could talk to each other, the final allocation would always be acceptable and *Pareto-optimal*: i.e., no agent could improve its utility without another agent decreasing its own utility. In our mechanism, however, participants only exchange load with their direct partners and this property does not hold. Instead, for a given load, the bounded-price mechanism limits the maximum difference in load levels that can exist between participants once the system converges to a final allocation. If a node has at least one task for which the unit marginal cost is greater than the upper-bound price of any of the node's contracts, then all its partners must have a load level such that an additional task would have an average unit marginal cost greater than the upper-bound price in their contract with the overloaded node. If a partner had a lower marginal cost, it would accept its partner's excess task at the contracted price. This property and the computation of the minimal price ranges yield the following theorem:

Theorem 2 *If nodes, contracts and tasks are homogeneous, and contracts are set according to Lemma 1, the final allocation is an acceptable allocation for underloaded systems and a nearly acceptable allocation for overloaded systems.*

In Section 5, we analyze heterogeneous settings using simulations and find that in practice, nearly acceptable allocations are also reached in such configurations.

Another property of the fixed-price mechanism is its fast convergence and low communication overhead. Task selection is the most complex operation and is performed once for each load offer and once for each response. Therefore, in a system with αN overloaded participants, under constant load, in the best case, all excess tasks require a single offer and are moved in parallel, for a convergence time of $O(1)$. In the worst case, the overloaded participants form a chain with only the last participant in the chain having spare capacity. In this configuration, participants must shed load one at the time, through the most ex-

[3]We exclude the task selection problem from the strategy space.

pensive of their C contracts, for a worst case convergence time of $O(NC)$. If nodes use notifications when they fail to shed load, the worst-case is reduced to $O(N+C)$. For auctions, the worst-case convergence time is $O(N)$ in this configuration. To summarize:

Lemma 3 *For N nodes with at most C contracts each, the fixed-price mechanism has a convergence time of $O(1)$ in the best case and $O(N+C)$ in the worst case if notifications are used.*

With our mechanism, most load movements require as few as three messages: an offer, a response, and a load movement. The best-case communication complexity is thus $O(N)$. In contrast, an approach based on auctions has a best-case communication overhead of $O(NC)$ if the auction is limited to C partners and $O(N^2)$ if not. If the whole system is overloaded and participants must move load through a chain of most-expensive contracts, the worst-case complexity may be as high as $O(N^2C)$, mostly because each participant periodically tries to shed load. If, however, the notifications described above are used, the worst-case communication overhead is only $O(NC)$. The same worst-case communication overhead applies to auctions. In summary:

Lemma 4 *To converge, the fixed-price contracts mechanism imposes a best-case communication overhead of $O(N)$ and a worst-case overhead of $O(NC)$ if notifications are used.*

Compared to auctions, our scheme significantly reduces the communication overhead, for a slight increase in worst-case convergence time. Simulation results (Section 5) show that the convergence time is short in practice.

When contracts specify a small price-range, most load movements take place outside of the range and the mechanism preserves the properties above. If, however, the optimal load allocation falls exactly within the small price-range, final convergence steps require more communication and may take longer to achieve, but at that point, the allocations are already acceptable. When load is forwarded through a chain, the complexity grows with the length of the chain and the amount of load that needs to be forwarded through that chain. In practice, though, long chains of overloaded nodes are a pathological, thus rare, configuration.

4 System Implementation

In this section, we describe the implementation of the bounded-price mechanism in the Medusa distributed stream-processing system.

4.1 Streams and Operators

In stream-processing applications, *data streams* produced by sensors or other data sources are correlated and aggregated by *operators* to produce outputs of interest. A data stream is a continuous sequence of attribute-value tuples that conform to some pre-defined schema. Operators are functions that transform one or more input streams into one or more output streams. A loop-free, directed graph of operators is called a *continuous query*, because it continuously processes tuples pushed on its input streams.

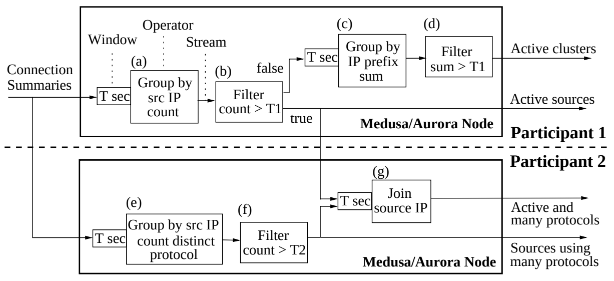


Figure 6: Example of a distributed Medusa query.

Stream processing applications are naturally distributed. Many applications including traffic management and financial feed analysis process data from either geographically distributed sources or different autonomous organizations. Medusa uses Aurora [1] as its query processor. Medusa takes Aurora queries and arranges for them to be distributed across nodes and participants, routing tuples and results as appropriate.

Figure 6 shows an example of a Medusa/Aurora query. The query, inspired from Snort [36] and Autofocus [11], is a simple network intrusion detection query. Tuples on input streams summarize one network connection each: source and destination IPs, time, protocol used, etc. The query identifies sources that are either active (operators `a` and `b`) or used abnormally large numbers of protocols within a short time period (operators `e` and `f`) or both (operator `g`). The query also identifies clusters of active sources (operators `c` and `d`). To count the number of connections or protocols the query applies *windowed aggregate* operators (`a`, `c`, and `e`): these operators buffer connection information for a time period T, group the information by source IP and apply the desired aggregate function. Aggregate values are then *filtered* to identify the desired type of connections. Finally, operator `g` *joins* active sources with those using many protocols to identify sources that belong in both categories.

The phrases in italics in the previous paragraph correspond to well-defined operators. While the system allows user-defined operators, our experience with a few applications suggests that developers will implement most application logic with built-in operators. In addition to simplifying application construction and providing query optimization opportunities, using standard operators facilitates Medusa's task movements between participants.

Medusa participants use *remote definitions* to move

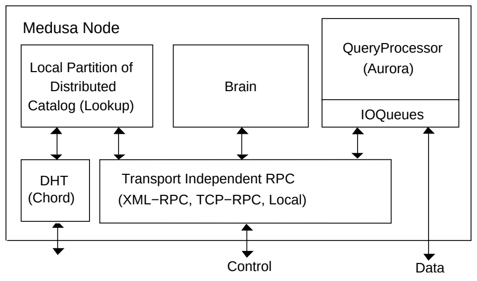

Medusa Node
Local Partition of Distributed Catalog (Lookup)
Brain
QueryProcessor (Aurora)
IOQueues
DHT (Chord)
Transport Independent RPC (XML−RPC, TCP−RPC, Local)
Control
Data

Figure 7: Medusa software structure.

tasks with relatively low overhead compared to full-blown process migration. Remote definitions specify how operators on different nodes map on to each other. At runtime, when a path of operators in the boxes-and-arrows diagram needs to be moved to another node, all that's required is for the operators to be instantiated remotely and for the incoming streams to be diverted to the appropriately named inputs on the new node. Our current prototype does not move operator state. The new instance re-starts the computation from an empty state. We plan to explore moving operator state in future work.[4]

For stream-processing, the algorithm for selecting tasks to offload to another participant must take into account the data flow between operators. It is preferable to cluster and move together operators that are linked with a high-rate stream or even simply belong to the same continuous query. In this paper we investigate the general federated load management problem and do not take advantage of possible relations between tasks to optimize their placement. In stream processing it is also often possible to partition input streams [41] and by doing so handle the load increase of a query network as a new query network. We make this assumption in this paper.

4.2 Medusa Software Architecture

Figure 7 shows the Medusa software structure. Each Medusa node runs one instance of this software. There are two components in addition to the Aurora query processor. The first component, called `Lookup`, is a client of an inter-node distributed catalog (implemented using a distributed hash table [42]) that holds information on streams, schemas and queries running in the system. The `Brain` component monitors local load conditions by periodically asking the `QueryProcessor` for the average input and output rates measured by the `IOQueues` (which serve to send and receive tuples to and from clients or other Medusa nodes) as well as for rough estimates of the local CPU utilization of the various operators. `Brain` uses this load information as input to the bounded-price mechanism that manages load.

[4]The semantics of many stream processing applications are such that occasional tuple drops are acceptable [2].

min # of contracts	1	3	5	7	9	10
max # of contracts	11	13	14	15	17	18
avg diameter	19	8	6	5	4	4

Table 1: Max number of contracts per node and network diameter for increasing min numbers of contracts.

5 Evaluation

In previous sections, we showed some properties of our mechanism and computed the best- and worst-case complexities. In this section, we complete the evaluation using simulations and experiments. We first examine the convergence to acceptable allocations in a network of heterogeneous nodes. We simulate heterogeneity by setting contracts at randomly selected prices. Second, we study the average convergence speed in large and randomly created topologies. Third, we examine how our approach to load management reacts to load variations. Finally, we examine how Medusa performs on a real application by running the network intrusion detection query, presented in Section 4, on logs of network connections.

We use the CSIM discrete event simulator [26] to study a 995-node Medusa network. We simulate various random topologies, increasing the minimum number of contracts per node, which has the effect of reducing the diameter of the contract network. To create a contract network, each node first establishes a bilateral contract with an already connected node, forming a tree. Nodes that still have too-few contracts then randomly select additional partners. With this algorithm, the difference between the numbers of contracts that nodes have is small, as shown in Table 1.

Each node runs a set of independent and identical operators that process tuples at a cost of 50 μs/tuple. We set the input rate on each stream to 500 tuples/s (or 500 KBps). Assuming that each node has one 100 Mbps output link, each operator uses 4% of the bandwidth and 2.5% of CPU. We select these values to model a reasonable minimum migratable unit. In practice, tasks are not uniform and the amount of resources each task consumes bounds how close a participant's marginal cost can get to a contract price without crossing it. When we examine convergence properties, we measure overload and available capacity in terms of the number of tasks that can be offered or accepted, rather than exact costs or load. All nodes use the same convex cost function: the total number of in-flight tuples (tuples being processed and tuples awaiting processing in queues).

5.1 Convergence to Acceptable Allocations

In this section, we study load distribution properties for networks of heterogeneous contracts. We compare the results to those achieved using homogeneous contracts and show that our approach works well in heterogeneous systems. We also simulate fixed-price contracts and show

Contracts	Price Selection
Fixed	$\forall i \in S\ :\ \lvert\text{taskset}_\text{i}^\text{F}\rvert = 14$ operators $\forall i,j \in S\ :\ \text{price}(C_{i,j}) = \text{MC}(u, \text{taskset}_\text{i}^\text{F} - u)$
Range	$\forall i,j \in S\ :$ $\text{max\_price}(C_{i,j}) = \text{MC}(u, \text{taskset}_\text{i}^\text{F} - u)$ $\text{min\_price}(C_{i,j}) = \text{max\_price}(C_{i,j}) - \Delta$ $\Delta = \delta_2(\text{taskset}_\text{i}^\text{F})$ i.e., the range is [MC 12th op,MC 14th op]
Random	$\forall i \in S\ :\ \lvert\text{taskset}_\text{i}^\text{F}\rvert \in [12\text{ops}, 18\text{ops}]$ $\forall i,j \in S$ if $\lvert\text{taskset}_\text{i}^\text{F}\rvert \leq \lvert\text{taskset}_\text{j}^\text{F}\rvert$ $\text{price}(C_{i,j}) = \text{MC}(u, \text{taskset}_\text{i}^\text{F} - u)$
Random Range	$\forall i,j \in S\ :$ max price same as for Random $\text{min\_price}(C_{i,j}) = \text{max\_price}(C_{i,j}) - \Delta$ $\Delta = \delta_2(\text{taskset}_\text{i}^\text{F})$

Desired result	Initial Load Allocation
underload	299 nodes with 22 ops 696 nodes with 7.7 ops on avg Overall average 12 ops/node
overload	299 nodes with 22 ops 696 nodes with 12.0 ops on avg Overall average 15 ops/node

Table 2: Simulated configurations. u is one operator. $\delta_2(\text{taskset}^\text{F})$ is defined in Section 3.2.3.

that such contracts have good properties in practice, even though they do not always lead to acceptable allocations.

We thus study and compare the convergence properties of four variants of the mechanism: (1) Fixed, where all contracts set an identical fixed-price, (2) Range, where all contracts are uniform but define a small price-range, (3) Random Fixed, where contracts specify fixed but randomly chosen prices, and Random Range, where contracts define randomly selected price-ranges. Table 2 summarizes the price selection used for each type of contract. We limit price ranges to the variation in marginal cost resulting from moving only two tasks. This range is much smaller than required in theory to ensure acceptable allocations: the range is half the theoretical value for the smallest network diameter that we simulate (Table 1).

Starting from skewed load assignments, as summarized in Table 2, we examine how far the final allocation is from being acceptable. For an underloaded system, we measure the total excess load that nodes were unable to move (Figure 8). For an overloaded system, we measure the unused capacity that remains at different nodes (Figure 9)[5]. In both figures, the column with zero minimum number of contracts shows the excess load and available capacity that exist before any reallocation.

With the Fixed variant of the mechanism, we examine the properties of an underloaded system whose fixed contract prices are above the average load level and those of an overloaded system where prices are below average load. As shown in Figures 8 and 9, as the number of contracts increases, the quality of the final allocation improves: nodes manage to reallocate a larger amount of

[5]Each result is the average of nine simulations.

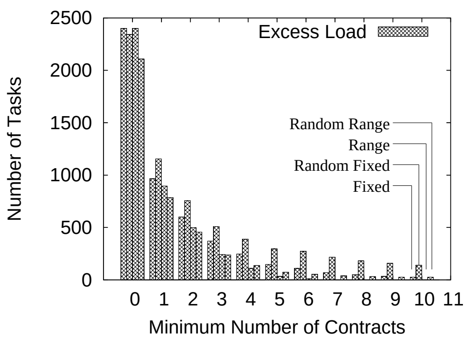


Figure 8: Excess load for the final allocation in an underloaded network of 995 nodes. First column shows excess load before any load movement.

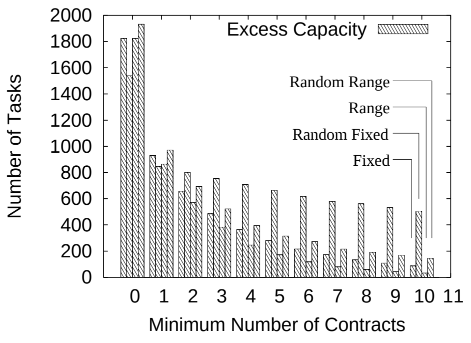
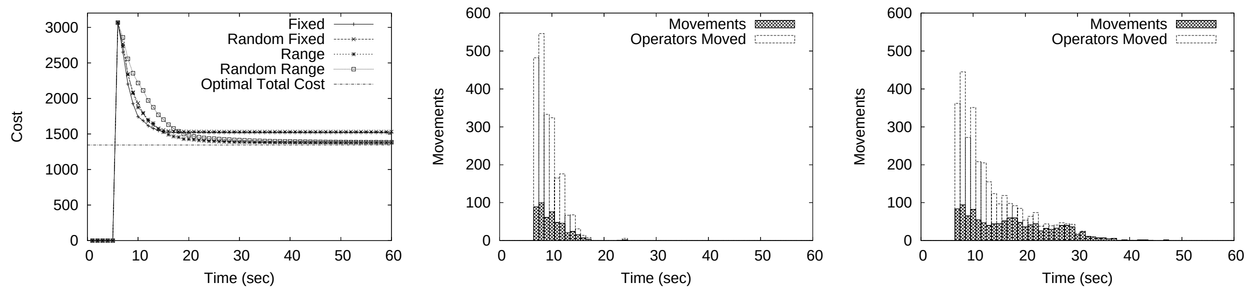


Figure 9: Left-over capacity for the final allocation in an overloaded network of 995 nodes.

excess load or exploit a larger amount of available capacity. With a minimum of 10 contracts, nodes successfully redistribute over 99% of excess load in the underloaded scenario and use over 95% of the initially available capacity in the overloaded case. The system thus converges to an allocation close to acceptable in both cases. Hence, even though they do not work well for specific configurations (as discussed in Section 3.2.3), fixed-price contracts can lead to allocations close to acceptable for randomly generated configurations.

We re-run all simulations replacing fixed prices with a price range and observe the improvement in final allocation. We choose the price range to fall within the two load levels of 12 and 15 operators per node (Table 2, variant Range). The results, also presented in Figures 8 and 9, show that a minimum of seven contracts achieves an acceptable allocation. At that moment, the diameter of the network is five (Table 1) so the price-range is half the theoretically required value. When the system is overloaded, over 98% of available capacity is successfully used with a minimum of 10 contracts per node. Additionally, nodes below their capacity have room for only one additional operator. The final allocation is thus nearly acceptable, as defined in Section 3.2.3. This result shows that price



(a) Total cost.

(b) Number of movements with fixed-price contracts.

(c) Number of movements with price-range contracts.

Figure 10: Convergence speed in an underloaded network of 995 nodes.

ranges, even smaller than the theoretically computed values, suffice to achieve acceptable or nearly acceptable allocations in randomly generated configurations.

In practice, contracted prices will be heterogeneous. We thus simulate the same network of contracts again but with randomly selected prices ($\mathrm{Random\ Fixed}$ variant). To pick prices, each participant randomly selects a maximum capacity between 12 and 18 operators. When two participants establish a contract, they use the lower of their maximums as the fixed price. We find that random fixed-price contracts are less efficient than homogeneous contracts but they also achieve allocations close to acceptable (94% of excess load is re-allocated with 10 contracts as shown in Figure 8). Because we measure the capacity at each node as the number of operators that the node can accept before its marginal cost reaches its highest contracted price, when the number of contracts increases, so does the measured capacity. This in turn makes heterogeneous-price contracts appear less efficient than they actually are at using available capacity. The range of prices from which contracts are drawn is such that when each node has at least three contracts, the initially available capacity is roughly the same as with homogeneous contracts.

Finally, we explore heterogeneous price ranges by adding a lower price bound to each randomly chosen fixed price ($\mathrm{Random\ Range}$ variant). As shown in Figures 8 and 9, heterogeneous price-range contracts have similar properties to the uniform price-ranges. The final allocation is slightly worse than in the uniform case because nodes with small capacity impede load movements through chains and measured capacity increases with the number of contracts. We find, however, that in the underloaded case nodes were at most 2.1 operators above their threshold (average of multiple runs) and in the underloaded case nodes had at most capacity left for 2.5 operators. Heterogeneous prices thus lead to allocations close to acceptable ones.

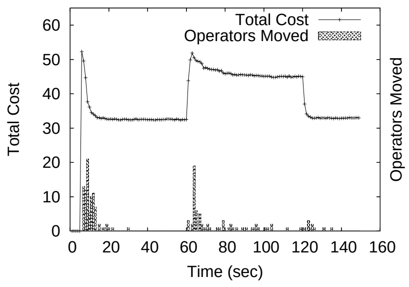
5.2 Convergence Speed

Figure 10(a) shows how the total cost decreases with time in the underloaded network of 995 nodes with a minimum of five contracts per node. In the simulation, each node tried to move load at most once every 2 seconds (no movements were allowed within the first 5 seconds of the simulation). For all variants, the cost decreases quickly and reaches values close to the final minimum within as few as 15 seconds of the first load movement. This fast convergence is partly explained by the ability of the bounded-price mechanism to balance load between any pair of nodes in a single iteration and partly by the ability of our mechanism to balance load simultaneously at many locations in the network. Fixed-price variants have a somewhat sharper decrease while also leading to a final allocation close to acceptable.

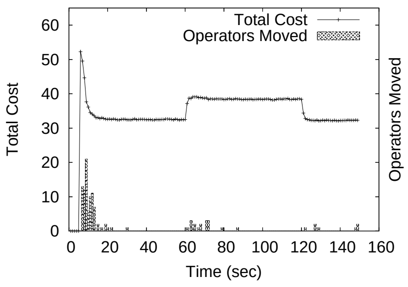
Figures 10(b) and (c) show the convergence speed measured as the number of load transfers ($\mathrm{Movements}$) and the number of operators moved for homogeneous contracts (random prices produce almost identical trends). For both variants, the first load movements re-allocate many operators, leading to the fast decrease in the total cost in early phases of the convergence. Fixed-price contracts converge faster than price-range contracts because more operators can be moved at once. The convergence also stops more quickly but, as shown above, the allocation is slightly worse than with a small price range.

5.3 Stability under Changing Load

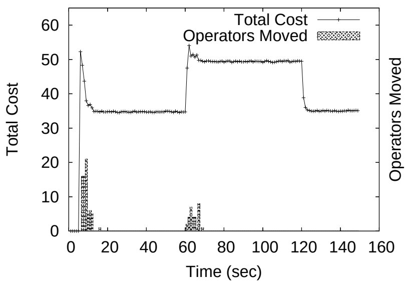
Next, we examine how the mechanism responds to sudden step-shifts in load. A bad property would be for small step shifts in load to lead to excessive re-allocations. We subject a network of 50 nodes (with a minimum of three fixed or three price-range contracts per node) to a sudden load increase (at time 60 sec) and a sudden load decrease (at time 120 sec). We run two series of experiments. We first create a large variation with 30% extra



(a) Large load variation with price range: 30% extra load.



(b) Small load variation with price range: 15% extra load.



(c) Large load variation with fixed prices: 30% extra load.

Figure 11: Assignment stability under variable load. Load added at 60 sec and removed at 120 sec.

load (10 extra operators each to 10 different nodes). We then repeat the experiment adding only 15% extra load. Figure 11 shows the total cost and operator movements registered during the simulation.

A 30% load increase, concentrated around a few nodes, makes these nodes exceed their capacity leading to a few re-allocations both with fixed prices and price ranges. Fixed prices, however, produce fewer re-allocations because convergence stops faster. When load is removed, spare capacity appears. If nodes use a price range, a small number of re-allocations follows. With fixed-prices, since nodes all run within capacity before the load is removed, nothing happens. The 15% load increase leads to an almost insignificant number of movements even when a small price-range is used. Indeed, fewer nodes exceed their capacity and load variations within capacity do not lead to re-allocations. Both variants of the mechanism thus handle load variations without excessive re-allocations.

5.4 Prototype Experiments

We evaluate our prototype on the network intrusion detection query (with 60 sec windows and without the final join) running on network connection traces collected at MIT (1 hour trace from June 12, 2003) and at an ISP in Utah (1 day trace from April 4, 2003). To reduce the possible granularity of load movements, we partition the Utah log into four traces that are streamed in parallel, and the MIT log into three traces that are streamed in parallel. To increase the magnitude of the load, we play the Utah trace with a 20$\times$ speed-up and the MIT trace with an 8$\times$ speed-up.

Figure 12 illustrates our experimental setup. Node 0 initially processes all partitions of the Utah and MIT traces. Nodes 1 and 2 process 2/3 and 1/3 of the MIT trace, respectively. Node 0 runs on a desktop with a Pentium(R) 4, 1.5GHz and 1GB of memory. Nodes 1 and 2 run on a Pentium III TabletPC with 1.33GHz and 1GB of memory. The nodes communicate over a 100 Mbps Ethernet. All clients are initially on the same machines as the nodes running their queries. All Medusa nodes have fixed-price contracts with each other and are configured to take or offer load every 10 seconds.

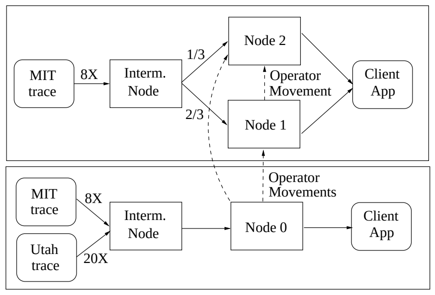


Figure 12: Experimental setup.

Figure 13 shows the results obtained. Initially, the load at each node is approximately constant. At around 650 seconds (1) the load on the Utah trace starts increasing and causes Node 0 to shed load to Node 1, twice (on Figure 13 these movements are labeled (2) and (3)). After the second movement, load increases slightly but Node 1 refuses additional load making Node 0 move some operators to Node 2 (4). The resulting load allocation is not uniform but it is acceptable. At around 800 seconds (5), Node 1 experiences a load spike, caused by an increase in load on the MIT trace. The spike is long enough to cause a load movement from Node 1 to Node 2 (6), making all nodes operate within capacity again. Interestingly, after the movement the load on Node 1 decreases. This decrease does not cause further re-allocations as the allocation remains acceptable.

In our experimental setup, it takes approximately 75 ms to move a query fragment between two nodes. Each movement proceeds as follows. The origin node sends to the remote node a list of operators and stream subscrip-

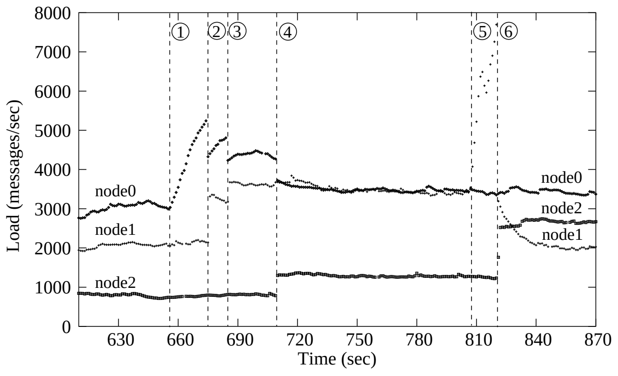


Figure 13: Load at three Medusa nodes running the network intrusion detection query over network connection traces.

tions (i.e., a list of client applications or other Medusa nodes currently receiving the query output streams). The remote node instantiates the operators locally, subscribes itself to the query input streams, starts the query, and sets up the subscriptions to the output streams. After the remote query starts, the origin node drains accumulated tuples and deletes the query. Both nodes update the catalog asynchronously. When a query moves, client applications see a small number of duplicate tuples because the new query starts before the old one stops. They may also see some reordering if the origin node was running behind before the move.

In our current implementation, we do not send the state of operators to the remote location. This approach works well for all stateless operators such as *filter*, *map*, and *union* as well as for operators that process windows of tuples without keeping state between windows (e.g., *windowed joins* and some types of *aggregates*). For these latter operators, a movement disrupts the computation over only one window. For more stateful operators, we should extend the movement protocol to include freezing the state of the original query, transferring the query with that state, and re-starting the query from the state at the new location. We plan to explore the movements of stateful operators in future work.

6 Conclusion

In this paper, we presented a mechanism for load management in loosely coupled, federated distributed systems. The mechanism, called the *bounded-price mechanism*, is based on pairwise contracts negotiated offline between participants. These contracts specify a bounded range of unit prices for load transfers between partners. At runtime, participants use these contracts to transfer excess load at a price within the pre-defined range.

Small price-ranges are sufficient for the mechanism to produce acceptable allocations in an underloaded network of uniform nodes and contracts, and produce allocations close to acceptable in other cases. Compared to previous approaches, this mechanism gives participants control and privacy in their interactions with others. Contracts allow participants not only to constrain prices but also practice price discrimination and service customization. The approach also has a low runtime overhead.

Participants have flexibility in choosing contract prices. We show that even randomly chosen prices from a wide range achieve allocations close to acceptable. We suggest, however, that participants first negotiate relatively high fixed-price contracts to maximize their chances of shedding excess load while minimizing runtime overhead and only later negotiate additional contracts with lower prices. Additionally, if participants notice that they often stand between overloaded and underloaded partners, they should re-negotiate some of their contracts to cover a small price-range and make a small profit by forwarding load from their overloaded to their underloaded partners.

Although the load management mechanism introduced in this paper is motivated by federated distributed stream processing, it also applies to other federated systems such as Web services, computational grids, overlay-based computing platforms, and peer-to-peer systems.

In this paper, we did not address high availability. Because each participant owns multiple machines, participant failures are rare. We envision, however, that if the participant running the tasks fails, it is up to the original nodes to recover the failed tasks. If the original participant fails, though, the partner continues processing the tasks until the original participant recovers. Contracts could also specify availability clauses. We plan to investigate high availability further in future work.

Acknowledgments

We thank Rahul Sami for many invaluable suggestions. We thank Jaeyeon Jung and Dave Andersen from MIT, and Noah Case from Syptec, for providing us the network connection traces.

References

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal: The Int. Journal on Very Large Data Bases*, Sept. 2003.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of 2002 ACM Symposium on Principles of Database Systems*, June 2002.

[3] P. Bhoj, S. Singhal, and S. Chutani. SLA management in federated environments. Technical Report HPL-98-203, Hewlett-Packard Company, 1998.

[4] R. Buuya, H. Stockinger, J. Giddy, and D. Abramson. Economic models for management of resources in peer-to-peer and grid computing. In *Proc. of SPIE Int. Symposium on The Convergence of Information Technologies and Communications (ITCom 2001)*, Aug. 2001.

[5] S. Chandrasekaran, A. Deshpande, M. Franklin, and J. Hellerstein. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, Jan. 2003.

[6] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for Internet databases. In *Proc. of the 2000 ACM SIGMOD Int. Conference on Management of Data*, May 2000.

[7] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, Jan. 2003.

[8] B. Chun, Y. Fu, and A. Vahdat. Bootstrapping a distributed computational economy with peer-to-peer bartering. In *Proc. of the Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[9] B. N. Chun. *Market-Based Cluster Resource Management*. PhD thesis, University of California at Berkeley, 2001.

[10] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, May 1986.

[11] C. Estan, S. Savage, and G. Varghese. Automatically inferring patterns of resource consumption in network traffic. In *Proc. of the ACM SIGCOMM 2003 Conference*, Aug. 2003.

[12] J. Feigenbaum, C. Papadimitriou, R. Sami, and S. Shenker. A BGP-based mechanism for lowest-cost routing. In *Proc. of the 21st Symposium on Principles of Distributed Computing*, July 2002.

[13] J. Feigenbaum, C. Papadimitriou, and S. Shenker. Sharing the cost of multicast transmissions. *Journal of Computer and System Sciences*, 63:21–41, 2001.

[14] J. Feigenbaum, C. Papadimitriou, and S. Shenker. Distributed algorithmic mechanism design: Recent results and future directions. In *Proc. of the 6th Int. Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, Sept. 2002.

[15] D. Ferguson, C. Nikolaou, J. Sairamesh, and Y. Yemini. Economic models for allocating resources in computer systems. In S. H. Clearwater, editor, *Market based Control of Distributed Systems*. World Scientist, Jan. 1996.

[16] I. T. Foster and C. Kesselman. Computational grids. In *Proc. of the Vector and Parallel Processing (VECPAR)*, June 2001.

[17] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An architecture for secure resource peering. In *19th ACM Symposium on Operating Systems Principles*, Oct. 2003.

[18] Y. Fu and A. Vahdat. Service level agreement based distributed resource allocation for streaming hosting systems. In *Proc. of 7th Int. Workshop on Web Content Caching and Distribution*, Aug. 2002.

[19] R. Gallager. A minimum delay routing algorithm using distributed computation. *IEEE Transactions on Communication*, COM-25(1), Jan. 1977.

[20] M. Jackson. Mechanism theory. *Forthcoming in Encyclopedia of Life Support Stystems*, 2000.

[21] A. Keller and H. Ludwig. The WSLA framework: Specifying and monitoring service level agreements for Web services. Technical Report RC22456, IBM Corporation, May 2002.

[22] J. F. Kurose. A microeconomic approach to optimal resource allocation in distributed computer systems. *IEEE Transactions on Computers*, 38(5):705–717, 1989.

[23] K. Lai, M. Feldman, I. Stoica, and J. Chuang. Incentives for cooperation in peer-to-peer networks. In *Proc. of the Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[24] W. Lehr and L. W. McKnight. Show me the money: Contracts and agents in service level agreement markets. `http://itc.mit.edu/itel/docs/2002/show_me_the_money.pdf`, 2002.

[25] T. W. Malone, R. E. Fikes, K. R. Grant, and M. T. Howard. Enterprise: A market-like task scheduler for distributed computing environments. *The Ecology of Computation*, 1988.

[26] Mesquite Software, Inc. CSIM 18 user guide. `http://www.mesquite.com`.

[27] M. S. Miller and K. E. Drexler. Markets and computation: Agoric open systems. In B. Huberman, editor, *The Ecology of Computation*. Science & Technology, 1988.

[28] C. Ng, D. C. Parkes, and M. Seltzer. Strategyproof computing: Systems infrastructures for self-interested parties. In *Proc. of the Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[29] C. Ng, D. C. Parkes, and M. Seltzer. Virtual worlds: Fast and strategyproof auctions for dynamic resource allocation. `http://www.eecs.harvard.edu/~parkes/pubs/virtual.pdf`, June 2003.

[30] T.-W. J. Ngan, D. S. Wallach, and P. Druschel. Enforcing fair sharing of peer-to-peer resources. In *Proc. of the 2nd Int. Workshop on Peer-to-Peer Systems (IPTPS '03)*, Feb. 2003.

[31] N. Nisan and A. Ronen. Computationally feasible VCG mechanisms. In *Proc. of the Second ACM Conference on Electronic Commerce (EC00)*, Oct. 2000.

[32] N. Nisan and A. Ronen. Algorithmic mechanism design. *Games and Economic Behavior*, 35, 2001.

[33] D. Parkes. *Iterative Combinatorial Auctions: Achieving Economic and Computational Efficiency (Chapter 2)*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 2001.

[34] D. Parkes. Price-based information certificates for minimal-revelation combinatorial auctions. *Agent Mediated Electronic Commerce IV*, LNAI 2531:103–122, 2002.

[35] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. In *Proc. of the First Workshop on Hot Topics in Networks*, Oct. 2002.

[36] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proc. of the 13th Conference on Systems Administration (LISA-99)*, Nov. 1999.

[37] A. Sahai, A. Durante, and V. Machiraju. Towards automated SLA management for Web services. Technical Report HPL-2001-310R1, Hewlett-Packard Company, July 2001.

[38] A. Sahai, S. Graupner, V. Machiraju, and A. van Moorsel. Specifying and monitoring guarantees in commercial grids through SLA. Technical Report HPL-2003-324, Hewlett-Packard Company, Nov. 2002.

[39] T. W. Sandholm. An implementation of the contract net protocol based on marginal cost calculations. In *Proc. of the 12th Int. Workshop on Distributed Artificial Intelligence*, pages 295–308, 1993.

[40] T. W. Sandholm. Contract types for satisficing task allocation: I theoretical results. In *AAAI Spring Symposium Series: Satisficing Models*, Mar. 1998.

[41] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proc. of the 19th Int. Conference on Data Engineering (ICDE 2003)*, Mar. 2003.

[42] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of the ACM SIGCOMM 2001 Conference*, pages 149–160, Aug. 2001.

[43] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: a wide-area distributed database system. *The VLDB Journal: The Int. Journal on Very Large Data Bases*, 5, Jan. 1996.

[44] The Condor Project. Condor high throughput computing. `http://www.cs.wisc.edu/condor/`.

[45] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer. KARMA: A secure economic framework for peer-to-peer resource sharing. In *Proc. of the Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[46] C. Waldspurger, T. Hogg, B. Huberman, J. Kephart, and W. Stornetta. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, SE-18(2):103–117, 1992.