Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

SEN

A system for distributed mechanisms: design, implementation and applications

K.R. Apt, F. Arbab, H. Ma

REPORT SEN-R0706 NOVEMBER 2007

Centrum voor Wiskunde en Informatica (CWI) is the national research institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organisation for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

A system for distributed mechanisms: design, implementation and applications

ABSTRACT
We describe here a structured system for distributed mechanism design. In our approach the players dynamically form a network in which they know neither their neighbours nor the size of the network and interact to jointly take decisions. The only assumption concerning the underlying communication layer is that for each pair of processes there is a path of neighbours connecting them.This allows us to deal with arbitrary network topologies. We also discuss the implementation of this system that consists of a sequence of layers. The lower layers deal with the operations relevant for distributed computing only, while the upper layers are concerned only with communication among players, including broadcasting and multicasting, and distributed decision making. This yields a highly flexible distributed system whose specific applications are realized as instances of a top layer. This design is implemented in Java. The system can be used for a repeated creation of dynamically formed networks of players interested in a joint decision making implemented by means of a tax-based mechanism. We illustrate its flexibility by discussing a number of implemented examples.

2000 Mathematics Subject Classification: 91A40, 91B26
1998 ACM Computing Classification System: C.2.4, J.4
Keywords and Phrases: mechanism design, Groves schemes, distributed implementation

A System for Distributed Mechanisms: Design, Implementation and Applications

Krzysztof R. Apt *† Farhad Arbab *‡ Huiye Ma *

November 16, 2007

Abstract

We describe here a structured system for distributed mechanism design. In our approach the players dynamically form a network in which they know neither their neighbours nor the size of the network and interact to jointly take decisions. The only assumption concerning the underlying communication layer is that for each pair of processes there is a path of neighbours connecting them. This allows us to deal with arbitrary network topologies.

We also discuss the implementation of this system that consists of a sequence of layers. The lower layers deal with the operations relevant for distributed computing only, while the upper layers are concerned only with communication among players, including broadcasting and multicasting, and distributed decision making. This yields a highly flexible distributed system whose specific applications are realized as instances of a top layer. This design is implemented in Java.

The system can be used for a repeated creation of dynamically formed networks of players interested in a joint decision making implemented by means of a tax-based mechanism. We illustrate its flexibility by discussing a number of implemented examples.

*CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
†University of Amsterdam
‡University of Leiden

1 Introduction

1.1 Background and motivation

Mechanism design is one of the important areas of economics. To quote from [6], it deals with the problem of 'how to arrange our economic interactions so that, when everyone behaves in a self-interested manner, the result is something we all like.' So these interactions are supposed to yield desired social decisions when each agent is interested in maximizing his utility.

Our interest here is in the distributed implementation of the large class of tax-based mechanisms that implement the decisions either in dominant strategies or in a Nash equilibrium (see, e.g., [9]). The traditional approaches rely on the existence of a central authority who collects the information from the players, computes the decision and informs the players about the outcome and their taxes.

Recently, in a series of papers distributed mechanism design was suggested as a realistic alternative for the applications based on the Internet. In this setting no central authority exists and the decisions are taken by the players themselves.

The challenge here is to appropriately combine the techniques of distributed computing with those that deal with the matters specific to mechanism design, notably rationality (i.e., appropriately defined self-interest) and truth-telling (i.e., incentive compatibility). We meet it by applying sound and proven techniques of distributed computing and, more generally, software engineering. More specifically, we propose an appropriate platform for distributed computing built out of a number of layers. This leads a flexible, hierarchical, design in which the lower layers are concerned with the communication and synchronization issues and are clearly separated from the upper layers that deal with the relevant aspects of the mechanism design.

More specifically, the lowest communication layer allows us to detect process failure and provides an asynchronous, non-order preserving `send` operation. The next layer provides a message efficient, fault-tolerant distributed termination detection (see, e.g., [10]) algorithm. In turn, the high-level communication layer provides primitives appropriate for communication among players, including broadcasting and multicasting. The dynamic network creation is realized by means of interconnected local registries.

Any specific application, such as an appropriate instance of the Groves mechanism (see, e.g., [9]), is realized as an instantiation of a top layer. Using this platform players can engage in joint decision making by dynamically forming a network with no central authority, in which they know neither their neighbours nor the size of the network.

This design is implemented in Java and was tested on a number of examples including Vickrey auction with redistribution, two types of auctions and a sequential mechanism design, described in the second part of the paper.

1.2 Related work

A number of recent papers deal with different aspects of distributed computing in connection with game theory and mechanism design.

Among them, Some focus on complexity such as communication complexity. Some target on computation/communication/incentive compatibility and eventually faithful implementation. Others try to build a secure computation in a distributed system. More recently, there has been a series of work on distributed constraint optimization and partial centralized technique.

[11] focused on message communication by players in a distributed game. However, they assume that there is a center to which every player is directly connected. An influential paper [5] introduced the notion of distributed algorithmic mechanism design emphasizing the issues of computational complexity and incentive compatibility in distributed computing. Next, [12] studied the distributed implementations of the VCG mechanism. However, in their approach there is still a center that is ultimately responsible for selecting and enforcing the outcome.

[17] considered the problem of creating distributed system specifications that will be faithfully implemented in networks with rational (i.e., self-interested) nodes so that no node will choose to deviate from the specification. They used interdomain routing as an example and suggested ways to detect when nodes deviate from their specified communication. In turn, [7] proposed in the context of secure computation a stronger form of computation in that it solely depends on players rationality not honesty.

[15] introduced the first distributed implementation of the VCG mechanism. The only central authority required was a bank that is in charge of the computation of taxes. The authors also discussed a method to redistribute some of the VCG payments back to players. Finally, [14] proposed a new partial centralization technique, PC-DPOP, based on the DPOP algorithm of [13]. PC-DPOP provides a better control over what parts of the problem

are centralized and allows this centralization to be optimal with respect to the chosen communication structure.

1.3 Details of our approach

Our work is closest to [15] whose approach is based on distributed constraint programming. In contrast, our approach builds upon a very general view of distributed programming, an area that developed a variety of techniques appropriate for the problem at hand.

This allowed us to realize the following improvements to the approach of [15]:

- we deal with a larger class of mechanisms, notably Groves mechanisms. They include the VCG mechanism and various forms of redistributions of VCG payments recently studied in the literature (and considered in [15]). Additionally, we can easily tailor our platform to other tax-based mechanisms, such as Walker mechanism (see [18]),
- the number of players can be unknown,
- the only central authority used is a ***tax authority***. It is weaker than the bank process of [15] in that it is needed only for the mechanisms that are not balanced, where it is used only to collect the resulting deficit,
- our platform can be easily customized to real-life applications by coupling it with specific registration schemes for participating in the mechanism. Also it can be used for a repeated distributed decision making process, each round involving a different group of interested players.

The lower layers of our platform support a generic ***broadcast command*** that ensures that each broadcast message is eventually delivered to each registered player. The implementation of this command relies only on the assumption that for each pair of players there is a path of neighbouring processes connecting them. This allows us to deal with arbitrary network topologies in a simple way.

Another distinctive feature of our approach is that it supports ***fault-tolerance*** at the mechanism design level. This means that the final decision and taxes can be computed even after some of the processes that broadcast the player's types crash: the other processes then still can proceed. This is achieved by the duplication of the computation by all players. Such a redundancy is common in all approaches to fault-tolerance. In [16] it is used to realize two natural requirements for a distributed mechanism implementation: computation compatibility and communication compatibility. Redundancy was intentionally avoided in [15] that aimed at minimizing the overall communication and computation costs. Here it allows the fastest process to 'dominate' the computation.

2 Mechanism design: the classical view

We recall here briefly tax-based mechanisms, notably the family of Groves mechanisms, see, e.g., [9, Chapter 23]. Assume a set of ***decisions*** D, a set $\{1, \ldots, n\}$ of players, for each player a set of ***types*** Θ_i and a ***utility function*** $v_i : D \times \Theta_i \rightarrow \mathcal{R}$. In this context a type is some private information known only to the player, for example a vector of player's valuations of the items for sale in a multi-unit auction.

A ***decision rule*** is a function $f : \Theta \rightarrow D$, where $\Theta := \Theta_1 \times \cdots \times \Theta_n$. We call the tuple

$$(D, \Theta_1, \ldots, \Theta_n, v_1, \ldots, v_n, f)$$

a ***decision problem***.

A decision rule f is called ***efficient*** if for all $\theta \in \Theta$ and $d' \in D$

$$\sum_{i=1}^{n} v_i(f(\theta), \theta_i) \geq \sum_{i=1}^{n} v_i(d', \theta_i),$$

and ***strategy-proof*** (or ***incentive compatible***) if for all $\theta \in \Theta$, $i \in \{1, \ldots, n\}$ and $\theta_i' \in \Theta_i$

$$v_i(f(\theta_i, \theta_{-i}), \theta_i) \geq v_i(f(\theta_i', \theta_{-i}), \theta_i),$$

where $\theta_{-i} := (\theta_1, \ldots, \theta_{i-1}, \theta_{i+1}, \ldots, \theta_n)$ and $(\theta_i', \theta_{-i}) := (\theta_1, \ldots, \theta_{i-1}, \theta_i', \theta_{i+1}, \ldots, \theta_n)$.

In mechanism design one is interested in the ways of inducing the players to announce their true types, i.e., in transforming the decision rules to the ones that are strategy-proof. In ***tax-based*** mechanisms this is achieved by extending the original decision rule by means of ***taxes*** that are computed by the central authority from the vector of the received types, using players' utility functions.

Given a decision problem, in the classical setting, one considers then the following sequence of events, where f is a given, publicly known, decision rule:

(i) each player i receives a type θ_i,

(ii) each player i announces to *the central authority* a type θ'_i; this yields a joint type $\theta' := (\theta'_1, \ldots, \theta'_n)$,

(iii) the central authority then makes the decision $d := f(\theta')$, computes the sequence of taxes $t := g(\theta')$, where $g : \Theta \rightarrow \mathcal{R}^n$ is a given function, and communicates to each player i the decision d and the tax $|t_i|$ he needs to pay to (if $t_i \leq 0$) or to receive from (if $t_i > 0$) the central authority.

(iv) the resulting utility for player i is then $u_i(d, t) := v_i(d, \theta_i) + t_i$.

Each ***Groves mechanism*** is obtained using $g(\theta') := (t_1(\theta'), \ldots, t_n(\theta'))$, where for all $i \in \{1, \ldots, n\}$

- $h_i : \Theta_{-i} \rightarrow \mathbb{R}$ is an arbitrary function,
- $t_i : \Theta \rightarrow \mathbb{R}$ is defined by[1]

$$t_i(\theta') := h_i(\theta'_{-i}) + \sum_{j \neq i} v_j(f(\theta'), \theta'_j).$$

Intuitively, the sum $\sum_{j \neq i} v_j(f(\theta'), \theta'_j)$ represents the society benefit from the decision $f(\theta')$, with player i excluded.

The importance of the Groves mechanisms is revealed by the following crucial result, in which we refer to the expanded decision rule $(f, g) : \Theta \rightarrow D \times \mathcal{R}^n$.

Groves Theorem Suppose the decision rule f is efficient. Then in each Groves mechanism the decision rule (f, g) is strategy-proof w.r.t. the utility functions $u_1, \ldots, u_n$.

The proof is remarkably straightforward so we reproduce it for the convenience of the reader.

[1] Here and below $\sum_{j \neq i}$ is a shorthand for the summation over all $j \in \{1, \ldots, n\}$, $j \neq i$.

Proof. Since f is efficient, for all $\theta \in \Theta$, $i \in \{1, \ldots, n\}$ and $\theta'_i \in \Theta_i$ we have

$$\begin{aligned} u_i((f,t)(\theta_i, \theta_{-i}), \theta_i) &= \sum_{j=1}^{n} v_i(f(\theta_i, \theta_{-i}), \theta_i) + h_i(\theta_{-i}) \\ &\geq \sum_{j=1}^{n} v_i(f(\theta'_i, \theta_{-i}), \theta_i) + h_i(\theta_{-i}) \\ &= u_i((f,t)(\theta'_i, \theta_{-i}), \theta_i). \end{aligned}$$

□

When for a given tax-based mechanism for all θ' we have $\sum_{i=1}^{n} t_i(\theta') \leq 0$, the mechanism is called ***feasible*** (which means that it can be realized without external financing) and when for all θ' we have $\sum_{i=1}^{n} t_i(\theta') = 0$, the mechanism is called ***balanced*** (which means that it can be realized without a deficit).

Each Groves mechanism depends on the functions $h_1, \ldots, h_n$. A special case, called ***Clarke mechanism***, or ***Vickrey-Clarke-Groves mechanism*** (in short ***VCG***) is obtained by using

$$h_i(\theta'_{-i}) := -\max_{d \in D} \sum_{j \neq i} v_j(d, \theta'_j).$$

So then

$$t_i(\theta') := \sum_{j \neq i} v_j(f(\theta'), \theta'_j) - \max_{d \in D} \sum_{j \neq i} v_j(d, \theta'_j).$$

Hence for all θ' and $i \in \{1, \ldots, n\}$ we have $t_i(\theta') \leq 0$, which means that the VCG mechanism is feasible and that each player needs to make the payment $|t_i(\theta')|$ to the central authority. Other feasible Groves mechanisms exist in which some players receive the payments and others have to make the payments, for example the one proposed in [4], which we discuss in Subsection 5.1.

3 Our approach

In our approach we relax a number of the assumptions made when introducing mechanism design. More specifically we assume that

- there is no central authority,
- players interested in 'the game' need to register,
- the players whose registration is accepted inform other registered players about their types,
- once a registered player learns that he has received the types from all registered players, he computes the decision and the taxes, sends this information to other registered players and terminates his computation.

We also assume that there is no collusion among the players. This leads to an implementation of the mechanism design by means of anonymous (i.e., name independent) distributed processes, in absence of any central authority. Because of the distributed nature of this approach no global state, in particular no global clock, exists. The computation of the decision and of the taxes is carried out by the players themselves.

As it stands, this revised setting is not clear on a number of counts. First, we need to clarify the registration process, in particular what it implies and when it ends. In our approach each player is represented by a process, in short a ***player process***. A player who wishes to join a specific mechanism (e.g., an auction) must register with a ***local registry***. Each geographic or logical region, such as a country, city, or Internet domain can have its own local registry. Players can find the addresses of their respective local registries in public fora, e.g., local government web sites. Local registries are linked together in a network that satisfies the full reachability condition described in Subsection 4.2 (and we assume one of them is designated as the initiator mentioned in that subsection). Receiving his registration request, a local registry verifies the eligibility of a player (e.g., whether his IP address puts him under the jurisdiction of this registry) and accepts his request if the registration conditions for the specific mechanism (e.g., a deadline) are met.

Second, once the registration process ends, in the resulting network a player process may not know the identities of other player processes, so the announcement of one's type to all other players needs to be explained. In our approach we assume that once a player process is registered, it joins the network of (registry and player) processes wherein a generic ***broadcast*** command is available. The topology of this network is irrelevant both from the point of view of the individual processes, as well as the semantics of the broadcast command. The full reachability of the backbone network of local registries is enough to ensure that as long as each player process knows and is known by its local registry, full reachability also holds for the whole network. The broadcast command uses the connectivity of this network to ensure that a copy of a broadcast message is eventually delivered to every registered player in finite time.

Third, we need to clarify how each player process will know that he indeed received the types announced by *all other* registered players. We solve this problem by assuming that each player process after broadcasting the player's type invokes a ***distributed termination detection algorithm*** the aim of which is to learn whether all players have indeed broadcast their types. This algorithm is tailored to deal with the communication by means of multicasting (which subsumes broadcasting).

If this algorithm detects termination, the player process knows that he indeed received all types, and in particular can determine at this stage the number of players. From that moment on each player process uses the same naming scheme when referring to other player processes. This is ensured by a local scheme for generating globally unique player identifiers. More generally, we use the distributed termination detection algorithm to detect the end of each *phase* of the distributed computation: registration, type broadcast, etc., i.e., for ***barrier synchronization***, see, e.g., [1].

Fourth, to ensure the correctness of the above approach, it is crucial that each player process computes the same decision and the same information concerning taxes. The former is taken care of by the fact that each player process uses the same, publicly known, decision rule f that each player learns, for example from a public bulletin board, and that is used by the player process after its registration is accepted.

Further, each player process applies f to the same input θ' and computes *the same* ***tax scheme*** by which we mean a specific vector of payments $tax(t_1), \ldots tax(t_n)$ computed from the tax vector $(t_1, \ldots, t_n)$, where $tax(t_j)$ specifies the amount that player j has to pay to other players and possibly the tax authority from his tax t_j. All tax schemes $tax(t_1), \ldots tax(t_n)$ then determine 'who pays how much to whom'. In general most taxes equal 0, so we optimize the computation by generating ***reduced tax schemes*** in which only non-zero entries are listed and by multicasting them instead of broadcasting. Note also that to compute the taxes each player process needs to know the utility functions of other player processes.

Finally, it is important to note that our approach allows for a ***repeated mechanism***, that is several rounds of decision making can take place, by means of the same given mechanism, each time involving a possibly different group of players. To this end we need to logically separate each round of the mechanism. This is handled, again, using our distributed termination detection algorithm for barrier synchronization.

4 Implementation

Our distributed mechanism design system is implemented in Java. The implementation follows the guidelines explained in the previous section. Figure 1 shows the overall architecture of our system and the different layers of software used in its implementation. The implementation of the first two layers is about 9K lines of Java code. It was developed by Kees Blom and took about 2 man years. The remainder of the system is about 3.5K lines of Java code and was developed by Huiye Ma during the last 9 months. We also relied on software for message passing between internet-based parallel processes developed by Han Noot. Each entity in this architecture communicates, either through function calls or method invocations, *only* with its adjacent entities. Specific applications are realized by instantiating the crucial player process layer.

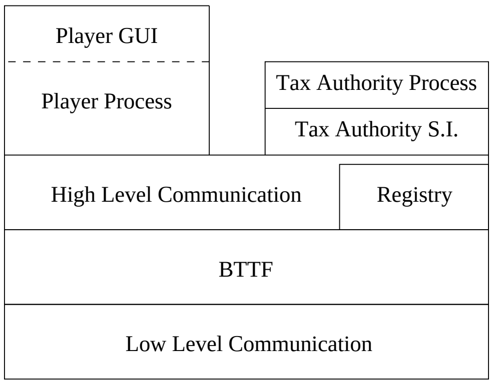

Figure 1: Implementation architecture

4.1 Low Level Communication

The Low Level Communication (LLC) layer supports (1) locally generated, globally unique process identifiers, and (2) reliable non-order-preserving, asynchronous, targeted communication, exclusively through the exchange of passive messages between processes. The only means of communication between processes in LLC is through message passing, where no transfer of control takes place when messages are exchanged.

Targeted means that the sender of a message explicitly specifies the recipient of the message. *Asynchronous* means that the receiver of a message is not guaranteed to have received the message upon the completion of the send operation. *Non-order-preserving* means that the temporal order of messages sent from the same sender to the same receiver is not necessarily preserved. *Reliable* means that every message sent by a sender will be received by its target receiver in finite but indeterminate time, without alteration and in its entirety (unless the receiver process fails or terminates). A message sent to a non-existent (terminated or failed) process will be returned to its sender intact, in finite but indeterminate time (a time-out).

The interface provided by the LLC layer contains the two operations `llsend(m, r)` and `llreceive(m, t)`. The `llsend(m, r)` operation sends the message `m` to its target process `r` and returns a Boolean value that indicates the success or failure of the operation. A send operation may fail, for instance, if the size of the message is above the capacity threshold of the transport mechanism, or due to other possible internal errors. Successful send simply means that the message has been dispatched on its way to its specified target.

The `llreceive(m, t)` operation blocks its calling process, p, until either (a) a message sent to p has arrived, or (b) the specified time-out `t` has expired. In the first case, `llreceive()` returns `true` and passes the received message in `m`. In the second case, this function returns `false` to indicate that the time-out `t` has expired. For convenience, we use the shorthand `llreceive(m)` for the common situation where the time-out `t` is infinity.

4.2 BTTF

The Back To The Future (BTTF) layer implements a message efficient, fault-tolerant distributed termination detection (DTD) algorithm, on top of the LLC layer. The details of the BTTF DTD algorithm are described in [3] and lie beyond the scope of this paper. We describe here only those salient features of this algorithm and its implementation that are pertinent for our application.

Specifically, the BTTF layer contains the implementation of the BTTF Wave algorithm, which is a wave DTD algorithm. All wave DTD algorithms determine termination using a cascading wave of special control messages, called tokens. They also require the designation of a single process as the ***initiator***, which is responsible for initiating the token waves, and typically, several rounds of token waves are necessary for the initiator to detect global termination. In each round, a cascading wave of tokens travels through every process in the system and collects its status information for the initiator.

In the BTTF algorithm, the initiator is anonymous, i.e., no process (other than the initiator) knows who the initiator is. All aspects of token handling and termination detection are transparently handled internally by the BTTF algorithm. The BTTF Wave algorithm is message efficient: in the absence of process failures, to detect termination in a system of m processes that exchange a total of n normal messages, it requires only $O(n)$ control messages plus 2 rounds of token waves, where each round contains between $O(m)$ to $O(m^2)$ token messages. The BTTF algorithm transparently detects and tolerates persistent process failures through an optional probing mechanism. Probing adds an extra cost of $O(n)$ control messages. Termination detection is costlier when process failures actually occur, because they may increase the number of required rounds of token waves: the BTTF Wave algorithm requires 2 successive failure-free rounds to detect termination.

Every process in the BTTF Wave algorithm must maintain a set of identifiers of $k \geq 0$ other processes in the system, called its *buddies*. The buddies sets of processes are transparently used by the BTTF Wave algorithm to cascade its token waves. The processes in the buddies set of a process p may or may not be the same as (some of) the processes that communicate with (i.e., send messages to, or receive messages from) p. The only requirement on the buddies sets of processes is that they must collectively provide the initiator with full reachability.

More precisely, let P be the finite set of processes in a system, and let $b^1(p)$ designate the buddies set of a process $p \in P$. For integers $i > 0$, define

$$b^{i+1}(p) = \bigcup_{x \in b^i(p)} b^i(x) \ \cup \ b^i(p).$$

Since P is finite, there exists an $i > 0$ for which the above definition reaches a fixed point $b^*(p)$, where $b^*(p) = b^{i+1}(p) = b^i(p)$. The full reachability requirement of the BTTF algorithm holds if for the initiator process $a \in P$, $b^*(a) \cup \{a\} = P$. Even when the failure of a process $x \in P$ partitions the remaining processes $P \setminus \{x\}$ into two or more mutually unreachable subsets, the BTTF algorithm continues undeterred in the partition $b^*(a) \cup \{a\} \setminus \{x\} \subset P \setminus \{x\}$ that includes the initiator process a. (In the calculation of $b^*(a)$, the $b^1(x)$ of a failed process x is $\emptyset$.)

In practice, there are many simple, local schemes that guarantee the full reachability requirement of the BTTF algorithm. For instance, in the common case where a system starts from a single process which transitively creates all other processes in the system, it is sufficient that each process keeps only its immediate parent and its immediate children in its buddies set. In this case, any process can be designated as the initiator, and each round of token waves involves $O(2m)$ token messages.

Aside from its responsibility to maintain its buddies list (e.g., adding its newly created children processes), a process using the BTTF algorithm is oblivious to the details of termination detection and failure recovery. A process starts by calling the initialization function provided by the BTTF layer. At this point the process is *active*. While active, a process can use the send and receive functions of the BTTF to send and receive messages to and from other processes. A process becomes *passive* when it is prepared to terminate. Termination is detected when all processes in the system are passive. It is possible for a process that is (still) active to send a message to a passive process, which must change the status of the receiver back to active, allowing it to send and receive more messages.

The BTTF layer provides two receive functions in its interface: `receive()` and `passiveReceive()`. A process uses `receive()` when it expects to receive a message from another process, while it is *not* prepared to terminate. A call to `receive()` blocks until it returns either with a received message, or when its optionally specified time-out parameter expires. Calling `passiveReceive()` indicates that the process is prepared to terminate, unless it receives a message. A call to `passiveReceive()` blocks until it returns either with a received message, or with an indication that global termination has been detected.

The DTD functionality provided by the BTTF layer can be used for barrier synchronization as well as for termination detection. Once `passiveReceive()` indicates termination has been detected, the calling process knows that all processes in the system have reached the same ‘termination barrier’. This termination barrier is either the actual termination of the processes, or the virtual termination of only the current phase of the activity in the system. In the first case, the calling process must perform its local clean-up and terminate. In the second case, the process must start a new *phase* of its computation by calling the initialization function of the BTTF layer once more.

The implementation of the BTTF layer requires only the `llsend(m, r)` and `llreceive(m, t)` operations provided by the LLC layer. It provides an interface consisting of the following functions:

- `initializeBTTFWave(...)` This function (re)initializes the calling process, enabling it to participate in the (next phase of) global computation. The details of the parameters of this function are beyond the scope of this paper.

- `insertBuddy(p)` This function call inserts the specified process `p` in the buddies set of the calling process.

- `removeBuddy(p)` This function call removes the specified process `p` from the buddies set of the calling process.

- `send(m, T)` This function implements a delayed multicast operation. It schedules a copy of the message `m` to be sent to every process in the target set of processes `T`. The actual dispatch of the messages to their specified targets will take place upon a subsequent call to one of the functions `prioritySend()`, `receive()`, or `passiveReceive()`.

- `prioritySend(m, T)` This function implements a multicast operation. It first sends all messages scheduled by earlier calls to `send()`, if any, and then sends a copy of the message `m` to every process in the target set of processes `T`.

- `receive(m, t)` The parameter `t` is an integer value. Negative `t` values indicate indefinite wait, and non-negative values specify a time-out value in milliseconds. A call to this function blocks until either the specified time-out expires, or a message sent to the calling process is available. If the specified time-out expires, the return result of this function is `false` and the value of `m` is undefined. If a received message is available, this function returns the message in `m` and returns `true`.

- `passiveReceive(m)` A call to this function blocks until either global termination (of the current phase of the computation) is detected, or a message sent to the calling process is available. If termination is detected, the return result of this function is `false` and the value of `m` is undefined. If a received message is available, this function returns the message in `m` and returns `true`.

4.3 High Level Communication and Registry

The High Level Communication (HLC) layer provides indirect, anonymous communication among the players in a distributed system. It includes a number of local registries whose mutual connectivity supports the full connectivity of the players necessary for broadcast. A player must sign-in at a local registry, after which it can use the other operations provided by the HLC layer to play the game. It provides the following functions:

- `signin(r)` This function signs the calling player process in at the local registry `r` and properly initializes the respective structures in both the registry `r` and the calling player process. The player can start the first phase of the game right after a successful return of a call to this function.

- `signout()` This function terminates the participation of the calling player process in the game.

- `bsend(m)` This function broadcasts the message `m` to all registered players in the game.

- `msend(m, T)` This function multicasts the message `m` to every player in the target set `T`.

- `receive(m, t)` This function is the same as its homonym in the BTTF layer.

- `passiveReceive(m)` This function is the same as its homonym in the BTTF layer.

4.4 Player Process

Specific applications are implemented using this top layer. It is built on top of the HLC layer and is used to implement specific actions of the players, in particular the computation of the decisions and taxes. In our implementation of the distributed mechanism design the following sequence of actions takes place for each player i, where $flag_i$ is a Boolean variable. By `termination loop` we mean here the statement

```
while (passiveReceive(m))
  {process message m}
```

and by `inspect loop` we mean the statement

```
flag = false
while (receive(m, 100)) {
  if (m is a tax scheme) {
    flag = true
    process m
  } else process or ignore m as appropriate
}
```

where 100 is some arbitrary time-out in miliseconds.

(i) process p_i representing player i is created and assigned a globally unique name,

(ii) p_i obtains player i's type,

(iii) p_i signs in at the local registry `r` in its region using the `signin(r)` call upon which all messages sent to p_i by processes representing other players are *locked* and stored. The lock prevents that p_i can access these messages,

(iv) if p_i receives the confirmation of the registration (the call of `signin(r)` is successful), it broadcasts player i's type using the `bsend()` function (and otherwise it terminates),

(v) the lock of p_i is open so that p_i can access all messages that were or will be sent to it by processes representing other players,

(vi) p_i performs the `termination loop`. When it ends (i.e., global termination is detected) p_i has a globally unique naming scheme at its disposal to refer to the processes that represent all registered players, and computes the number of players n from the number of types it has received,

(vii) p_i performs the `inspect loop` to determine whether another process has already computed player i's tax scheme. If this is the case, $flag_i$ will be set `true`,

(viii) if $flag_i$ is not `true`, p_i computes the decision and the tax schemes of the players and multicasts using the `msend()` function the decision and the tax schemes to the processes representing players who need to pay or receive taxes and the decision to the other processes. If p_i needs to pay some tax $t' > 0$ to the tax authority it sends this information to the tax authority process using the `msend()` function,

(ix) p_i performs the `termination loop`,

(x) when it ends and after p_i receives from the tax authority process the total amount of taxes the tax authority received, p_i performs the `termination loop` again and terminates.

The details of the tax scheme algorithm can be found in Appendix I.

4.5 Tax Authority Software Interface

This layer is built on top of the HLC layer. It provides two functions also available in the HLC layer, `passiveReceive(m)` and `bsend(m)`, and two new functions, `tsignin(r)` and `tsignout()`, which are the counterparts of the `signin(r)` and `signout()` functions of the HLC layer and which are used to deal with the tax authority process registration.

4.6 Tax Authority Process

This layer is built on top of the Tax Authority Software Interface layer and is used to implement the actions of the tax authority which is in charge of collecting players' taxes. The following sequence of actions takes place for it:

(i) The tax authority process *ta* representing the tax authority is created and assigned a globally unique name known to every player. It signs in at the local registry in his region using the `tsignin(r)` call (which always succeeds),

(ii) *ta* performs the `termination loop` (to synchronize the computation phases with the player processes),

(iii) *ta* performs the `termination loop` again. When it ends, the tax authority process has received all the taxes from the players. They are kept on a single account,

(iv) *ta* broadcasts the total amount on its single account to all players,

(v) *ta* performs the `termination loop` and terminates.

4.7 Player GUI

The interaction between the player (user) and the system is realized in this interface. The interaction is limited to the registration, type submission and tax reception.

4.8 Comments

The above approach can be easily customized to specific purposes. In particular we can add specific registration details, for example stipulate that the registration is successful only if it took place before a certain deadline that refers to a global clock, or if some quorum (minimum number) of registered players is reached at each local registry, and/or if a global quorum of registered players is reached.

As it stands our design seems to allow some players to modify the information (for example, a type or tax schemes) that is passed through them to other players or to the tax authority in ways that are advantageous for themselves.

However, the system is so designed that this information is available only *within* the player process layer *and not* to the players themselves. Indeed, the players access the system only through the player GUI that limits the interface between the players and the system to a minimum. This provides a form of security to the users of the systems.

Further, the use of locks in the player process layer provides a protection against tampering with this layer. However, no provision has been implemented against 'breaking' into lower layers of the system. These matters are orthogonal to the ones considered here and their solution would require use of specific techniques from computer security and cryptography.

5 Examples

We used our distributed mechanism design system in a number of test cases that we now briefly describe. Each of them, is implemented as an instantation of the player process layer described in Subsection 4.4.

5.1 Vickrey auction with redistribution

In Vickrey auction there is a single object for sale which is allocated to the highest bidder who pays the second highest bid. We consider here the proposal of [4] in which the highest bidder redistributes some amounts from his payment to other players. This minimizes the overall tax.

First we model Vickrey auction as the following decision problem $(D, \Theta_1, \ldots, \Theta_n, v_1, \ldots, v_n, f)$:

- $D = \{1, \ldots, n\}$,
- each Θ_i is the set $\mathcal{R}_+$ of non-negative reals; $\theta_i \in \Theta_i$ is player i's valuation of the object,
- $v_i(d, \theta_i) := \begin{cases} \theta_i & \text{if } d = i \\ 0 & \text{otherwise} \end{cases}$
- $f(\theta) := i$,
  where $\theta_i = \max_{j \in [1..n]} \theta_j$ and[2] $\forall j \in [i+1..n]\ \theta_j < \theta_i$.

Here decision $d \in D$ indicates to which player the object is sold. By definition f is an efficient decision rule. Below, given a sequence s of reals we denote by $[s]_k$ the kth largest element in this sequence. For example, for $\theta = (1, 5, 2, 3, 2)$ we have $[\theta_{-2}]_2 = 2$ since $\theta_{-2} = (1, 2, 3, 2)$.

[2] In case of a tie we allocate the object to the player with the highest index.

The payments (taxes) in Vickrey auction are realized by applying the VCG mechanism, which yields

$$t'_i(\theta) := \begin{cases} -[\theta]_2 & \text{if } f(\theta) = i \\ 0 & \text{otherwise} \end{cases}$$

To formalize the redistribution scheme of [4] in our framework we combine each tax t'_i with the following function h_i (to ensure that it is well-defined we need to assume that $n \geq 3$):

$$h_i(\theta_{-i}) := \frac{[\theta_{-i}]_2}{n}$$

that is, by using

$$t_i(\theta) := t'_i(\theta) + h_i(\theta_{-i}).$$

Note that this yields a Groves mechanism since by the definition of the VCG mechanism for specific functions $h'_1, \ldots, h'_n$

$$t'_i(\theta) := h'_i(\theta_{-i}) + \sum_{j \neq i} v_j(f(\theta), \theta_j)$$

and consequently

$$t_i(\theta) = (h_i + h'_i)(\theta_{-i}) + \sum_{j \neq i} v_j(f(\theta), \theta_j).$$

The resulting mechanism is feasible since for all $i \in [1..n]$ and θ we have $[\theta_{-i}]_2 \leq [\theta]_2$ and as a result

$$\sum_{i=1}^{n} t_i(\theta) = \sum_{i=1}^{n} t'_i(\theta) + \sum_{i=1}^{n} h_i(\theta_{-i}) = \sum_{i=1}^{n} \frac{-[\theta]_2 + [\theta_{-i}]_2}{n} \leq 0.$$

Let, given the sequence θ of submitted bids (types), π be the permutation of $1, \ldots, n$ such that $\theta_{\pi(i)} = [\theta]_i$ for $i \in [1..n]$ (where we break the ties by selecting players with the higher index first). So the ith highest bid is by player $\pi(i)$ and the object is sold to player $\pi(1)$. Then

- $[\theta_{-i}]_2 = [\theta]_3$ for $i \in \{\pi(1), \pi(2)\}$,
- $[\theta_{-i}]_2 = [\theta]_2$ for $i \in \{\pi(3), \ldots, \pi(n)\}$,

so the above mechanism boils down to the following payments by player $\pi(1)$:

- $\frac{[\theta]_3}{n}$ to player $\pi(2)$,
- $\frac{[\theta]_2}{n}$ to players $\pi(3), \ldots, \pi(n)$,
- $[\theta]_2 - \frac{2}{n}[\theta]_3 - \frac{n-2}{n}[\theta]_2 = \frac{2}{n}([\theta]_2 - [\theta]_3)$ to the tax authority,

that is, it does indeed coincide with the scheme of [4].

5.2 Unit demand auction

We now consider an auction with multiple items offered for sale. We assume that there are n players and m items and that each player submits a valuation for each item. The items should be allocated in such a way that each player receives at most one of them and the aggregated valuation is maximal.

This auction can be modelled as the following decision problem:

- $D = \{f \mid f : A \to \{1, \ldots, n\}, A \subseteq \{1, \ldots, m\}, f \text{ is 1-1}\}$,
  i.e., each decision is a 1-1 allocation of items to players,
- $\Theta_i = \mathcal{R}_+^m$; $(\theta_{i,1}, \ldots, \theta_{i,m}) \in \Theta_i$ is a vector of player i's valuations of the items for sale,
- $v_i(d, \theta_i) := \begin{cases} \theta_{i,j} & \text{if } d(j) = i \\ 0 & \text{if } \neg\exists j\ d(j) = i \end{cases}$
- $f(\theta') := d$ for which $\sum_{j \in dom(d)} \theta'_{d(j),j}$ is maximal.

Decision rule f is clearly efficient, so Groves Theorem can be used. Our distributed implementation of the corresponding VCG mechanism is again realized as an instance of the player process layer of Subsection 4.4 with the following details concerning computation of the decision and taxes.

When a player has received the types from all the registered players he needs to compute the decision. To this end we use the Kuhn-Munkres algorithm to compute the maximum weighted matching, where the weight associated with the edge (j, i) is the valuation for item j reported by player i. In our implementation we used the Java source code available at `http://adn.cn/blog/article.asp?id=49`.

To compute tax for player i according to the VCG mechanism this algorithm needs to be used again, to compute the maximum weighted matching with player i excluded.

5.3 Single minded auction

Next we consider an auction studied in [8] in which there are n players and m items, with each player only interested in a specific set of items (which explains the name of the auction). In our approach we limit ourselves to the situation in which each player i is only interested in a consecutive sequence $a_i, \ldots, b_i$ of the items $1, \ldots, m$, with $1 \leq a_i \leq b_i \leq m$.

We model this as the following decision problem:

- $D = \{f \mid f : A \rightarrow \{1, \ldots, n\},\ A \subseteq \{1, \ldots, m\}\}$,
- $\Theta_i = \mathcal{R}_+$; $\theta_i \in \Theta_i$ is player i's valuation for the sequence $a_i, \ldots, b_i$ of the items,
- $v_i(d, \theta_i) := \begin{cases} \theta_i & \text{if } d(j) = i \text{ for all } j \in [a_i, \ldots, b_i] \\ 0 & \text{otherwise} \end{cases}$
- $f(\theta') := d$ for which $\sum_{i:d([a_i,\ldots,b_i])=\{i\}} \theta'_i$ is maximal, where $d([a_i, \ldots, b_i]) = \{d(j) \mid j \in [a_i, \ldots, b_i]\}$.

So, given an allocation $f \in D$ the goods in the set $\{k \mid f(k) = j\}$ are allocated to player j. Note that alternatively f can be defined by:

$$f(\theta') := d \text{ for which } \sum_{i=1}^{n} v_i(d, \theta'_i) \text{ is maximal.}$$

So f is efficient and consequently Groves Theorem applies. The computations of the decision and of the taxes within the player process layer of Subsection 4.4 involve constructions of the maximum weighted matchings that are computed using a dynamic programming algorithm, details of which are omitted.

5.4 Sequential Groves mechanisms

In the original set up of the decision problem all players announce their types independently. In a modification studied in [2] the types are announced sequentially, in a random order.

Suppose that the random order is $1, \ldots, n$. The crucial difference between the customary set up and the one now considered is that player i knows the types announced by players $1, \ldots, i-1$. In [2] it was shown that in the context of Groves mechanisms used for problems concerned with public projects players have then other dominant strategies than truth-telling (i.e., announcing their true type) and that these strategies can be used to minimize the taxes.

Sequential Groves mechanisms can be implemented by means of our distributed mechanism system by slightly modifying the player process layer, specifically items (iv) and (v) in the sequence of actions described in Subsection 4.4 to:

- if process p_i receives the confirmation of the registration, it includes his sequence number j and information whether it represents the last player (the latter is needed to use other dominant strategies than truth-telling),
- the lock of p_i is partly open so that p_i can access all messages that were or will be sent to it by processes representing players with sequence number $< j$,
- each process with sequence number j, where $j > 1$, counts the number of types it received. When the count becomes $j - 1$ it broadcasts the type,
- the lock of p_i is then (completely) open.

5.5 Other applications

To test the versatility of our approach we also implemented a number of other examples. These include:

- Vickrey auction,
- a number of examples of decision making concerned with public projects, see [9, Chapter 23],
- Walker mechanism of [18].

In the latter mechanism each player i has a utility function of the form $v_i(q) := b_i(q) - c_i(q)$. Here q is the total amount of public good (for example grass area in a city) produced by the players, $b_i(q)$ is the benefit for player i from the amount of q of public good, and $c_i(q)$ is the cost share player i has to pay.

Each player i reports a real number x_i, which is interpreted as the amount of public good he agrees to produce. Then he receives the payment (tax)

$$t_i(x) := (x_{i+1} - x_{i-1}) \sum_{j=1}^{n} x_j,$$

where we interpret $n+1$ as 1 and $1-1$ as n, that is $i+1$ and $i-1$ are the indices of the right-hand and left-hand neighbours of player i in a ring.

So $x = \sum_{j=1}^{n} x_j$ is the total amount of public good produced and the final utility for player i is of the form $u_i(x) := v_i(x) + t_i(x)$.

This mechanism is not an instance of Groves mechanism and implements the decision not in dominant strategies but in a Nash equilibrium. To implement it we again merely modified the player process layer. To test this mechanism we used specific functions b_i and c_i.

6 Conclusions and future work

We presented in this paper a design of a platform that supports distributed mechanism design. It is built as a sequence of layers. The lower layers provide support for distributed computing, while the upper ones are concerned only with the matters specific to mechanism design. The platform is implemented in Java.

We believe that the proposed platform clarifies how the design of systems supporting distributed decision making can profit from sound principles of software engineering. We found that the division of the software into layers resulted in a flexible design that could be easily customized to specific mechanisms proposed in the literature, such as (sequential) Groves mechanisms and Walker mechanism, and to specific applications, such as various forms of auctions.

We also provided evidence that software engineering in the area of multiagent systems can profit from the techniques developed in the area of distributed computing, for example broadcasting in an environment with an unknown number of processes, distributed termination and barrier synchronization.

In our work we have not dealt with the problem of false-name bids, see [19], that needs to be addressed anew in the context of distributed implementations. This is the subject of our current research. Also, we plan to use our system to implement continuous double auctions.

Acknowledgments

Kees Blom kindly modified his implementation of the first two layers to our purposes. We acknowledge helpful discussions with Han Noot on the matter of using Java in a distributed environment. The applications to the unit demand and single minded auctions were suggested by Vangelis Markakis who also proposed the appropriate algorithms. The work of Huiye Ma was funded by the NWO project DIACoDeM, No 642.066.604.

References

[1] G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Systems*. Addison Wesley, New York, 2005.

[2] K. R. Apt and A. Estévez-Fernández. Sequential mechanism design. Manuscript, CWI, Amsterdam, The Netherlands, 2007. Available from `http://arxiv.org/abs/0705.2170`.

[3] F. Arbab. Back To The Future: A family of algorithms for termination detection in distributed systems. Technical report, CWI, Amsterdam, The Netherlands, 2007. In preparation.

[4] R. Cavallo. Optimal decision-making with minimal waste: Strategyproof redistribution of VCG payments. In *AAMAS '06: Proceedings 5th Int. Joint Conf. on Autonomous Agents and Multiagent Systems*, pages 882–889. ACM Press, 2006.

[5] J. Feigenbaum and S. Shenker. Distributed algorithmic mechanism design: recent results and future directions. In *DIALM '02: Proceedings of the 6th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, pages 1–13, New York, NY, USA, 2002. ACM Press.

[6] Intelligent design. The Economist, October 18th, 2007.

[7] S. Izmalkov, S. Micali, and M. Lepinski. Rational secure computation and ideal mechanism design. In *FOCS '05: Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, pages 585–595, Washington, DC, USA, 2005. IEEE Computer Society.

[8] D. Lehmann, L. O'Callaghan, and Y. Shoham. Truth revelation in approximately efficient combinatorial auctions. *Journal of the ACM*, 49(5):1–26, 2002.

[9] A. Mas-Collel, M. Whinston, and J. Green. *Microeconomic Theory*. Oxford University Press, 1995.

[10] J. Matocha and T. Camp. A taxonomy of distributed termination detection algorithms. *Journal of Systems and Software*, 43(3):207–221, 1998.

[11] D. Monderer and M. Tennenholtz. Distributed games: from mechanisms to protocols. In *AAAI '99/IAAI '99: Proceedings of the 16th National Conference on Artificial intelligence and 11th Conference on Innovative Applications of Artificial Intelligence*, pages 32–37, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence.

[12] D. C. Parkes and J. Shneidman. Distributed implementations of Vickrey-Clarke-Groves mechanisms. In *AAMAS '04: Proceedings 3rd Int. Joint Conf. on Autonomous Agents and Multiagent Systems*, pages 261–268, 2004.

[13] A. Petcu and B. Faltings. DPOP: A scalable method for multiagent constraint optimization. In *IJCAI '05: Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 266–271, Aug 2005.

[14] A. Petcu, B. Faltings, and R. Mailler. PC-DPOP: A new partial centralization algorithm for distributed optimization. In *IJCAI '07: Proceedings of the 20th International Joint Conference on Artificial Intelligence*, 2007.

[15] A. Petcu, B. Faltings, and D. C. Parkes. MDPOP: faithful distributed implementation of efficient social choice problems. In *AAMAS '06: Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1397–1404, New York, NY, USA, 2006. ACM Press.

[16] J. Shneidman and D. C. Parkes. Using redundancy to improve robustness of distributed mechanism implementations. In *EC '03: Proceedings*

of the 4th ACM conference on Electronic commerce, pages 276–277, New York, NY, USA, 2003. ACM Press.

[17] J. Shneidman and D. C. Parkes. Specification faithfulness in networks with rational nodes. In *PODC '04: Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, pages 88–97, New York, NY, USA, 2004. ACM Press.

[18] M. Walker. A simple incentive compatible scheme for attaining Lindahl allocations. *Econometrica*, 49(1):65–71, 1981.

[19] M. Yokoo. Pseudonymous bidding in combinatorial auctions. In P. Cramton, Y. Shoham, and R. Steinberg, editors, *Combinatorial Auctions*, pages 161–187. The MIT Press, 2006.

Appendix I

We explain here the details of the reduced tax scheme algorithm mentioned in Section 3. Intuitively, this algorithm determines given the tax vector $(t_1, \ldots, t_n)$ 'who pays how much to whom'.

We consider a list of players, each with his tax, and assume that the tax vector is feasible, that is the total sum of taxes is non-positive. This means that the claims of the players whose taxes are positive can be financed by the players whose taxes are negative.

First the players are divided into two lists, $A^0_{neg}, \ldots, A^k_{neg}$, consisting of players whose taxes are negative (i.e., those who should pay the taxes) and $A^0_{pos}, \ldots, A^m_{pos}$ consisting of players whose taxes are strictly positive (i.e., those who should be paid). Players whose tax is 0 are omitted.

We start with player A^0_{neg} and compare the absolute value of his tax, $|t^0_{neg}|$, with the tax t^0_{pos} of player A^0_{pos}.

If $|t^0_{neg}| \geq t^0_{pos}$, player A^0_{neg} pays the amount t^0_{pos} to player A^0_{pos}. This changes the tax of player A^0_{neg} from t^0_{neg} to $t^0_{neg} + t^0_{pos}$. The process is now repeated with player A^0_{neg} and the next unpaid player, A^1_{pos}.

If $|t^0_{neg}| < t^0_{pos}$, then player A^0_{neg} pays the amount $|t^0_{neg}|$ to player A^0_{pos}. This changes the tax of player A^0_{pos} from t^0_{pos} to $t^0_{pos} + t^0_{neg}$. The process is now repeated with the next player who should pay a tax, A^1_{neg}, and player A^0_{pos}.

The loop stops when all players with negative taxes paid. Termination is ensured by the assumption that the tax scheme is feasible. If the mechanism is not balanced, upon termination each player that still needs to pay some tax pays it to the tax authority.

The pseudo-code of the algorithm is given in Figure 2.

L_{all} is the list of n players;
A^i is the $(i+1)$st player in the list L_{all};
n_{all} is the length of the list L_{all};
t^i is the tax of player A^i;
tax is the list representing the computed tax scheme;
for $i = 0$ to n_{all} **do**
  if $t^i < 0$ **then**
    append A^i to the list L_{neg};
  end if
  if $t^i > 0$ **then**
    append A^i to the list L_{pos};
  end if
end for;
let A^j_{neg} be the $(j+1)$st player in the list L_{neg};
t^j_{neg} is the tax of player A^j_{neg};
let A^k_{pos} be the $(k+1)$st player in the list L_{pos};
t^k_{pos} is the tax of player A^k_{pos};
let n_{neg} be the length of the list L_{neg};
let n_{pos} be the length of the list L_{pos};
let t_{cursum} be the current sum of all the negative taxes not yet paid;
if n_{neg} != 0 **then**
  $k = 0$; $j = 1$; $t_{cursum} = t^0_{neg}$;
  while $j \leq n_{neg}$ and $k < n_{pos}$ **do**
    if $|t_{cursum}| \geq t^k_{pos}$ **then**
      player $j-1$ pays player k
      $amount = t^k_{pos} - (|t_{cursum}| - |t^{j-1}_{neg}|)$;
      $t^{j-1}_{neg} = t^{j-1}_{neg} + (t^k_{pos} - (|t_{cursum}| - |t^{j-1}_{neg}|))$;
      $t_{cursum} = t^{j-1}_{neg}$;
      $k = k + 1$;
      if $t_{cursum} == 0$ **then**
        $t_{cursum} = t^j_{neg}$; $j = j + 1$;
      end if
    else
      player $j-1$ pays player k $amount = |t^{j-1}_{neg}|$;
      $t_{cursum} = t_{cursum} + t^j_{neg}$; $j = j + 1$;
    end if
    $tax = tax + (j-1, k, amount)$;
  end while
end if

Figure 2: The algorithm to compute reduced tax scheme

Appendix II

In this appendix we illustrate a sample interaction with the platform. We assume that each player chooses from the pull down menu a single minded auction, discussed in Section 5.3. We consider a specific instance with

- 5 players,
- 3 items for sale,
- the following players bids: A: 20:(1,2), B: 50:(3), C: 32:(2), D: 60:(2,3), E: 19:(1),

  that is, player A bids 20 for the bundle (1,2), etc.

The registration process was taken care of by creating two local registries. In this example, the generated allocation is: (3:B, 28), (2:C, 10), (1:E, 0), that is item 3 is sold to player B who pays for it to the tax authority 28, etc.

The interaction with the system is presented in Figures 3 – 8 below. The first two figures depict phase 1 which consists of the registration process for players A and B. The 2nd phase, depicted in Figures 5 and 6, is type submission that takes place after the registration is accepted.

The 3rd phase consists of the computation of the tax scheme, its multicasting of it to other players and (in case of unbalanced mechanism) payment of the remaining taxes to the tax authority. The 4th phase consists of receiving by the players information from the tax authority about the overal tax received by it. These two phases are depicted in Figures 7 and 8. They show the difference in computation between fast players (here player A) and slow players (here player B). In this example, in phase 3, the tax scheme was only computed by the fast player, A, who subsequently multicast it.

Figure 3: Phase 1: player A

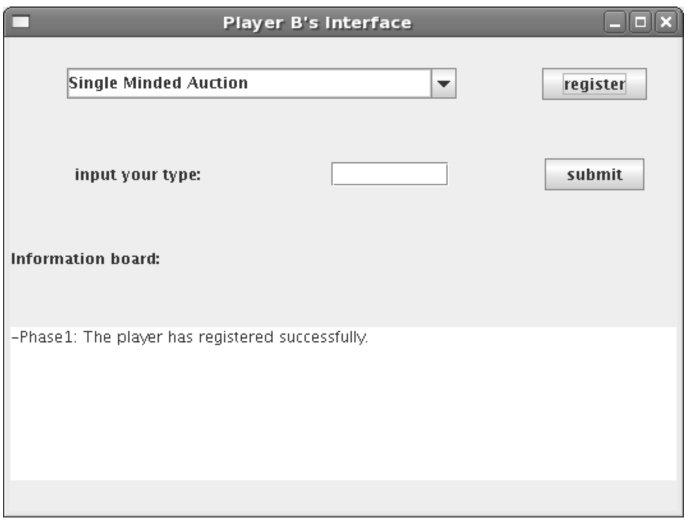

Figure 4: Phase 1: player B

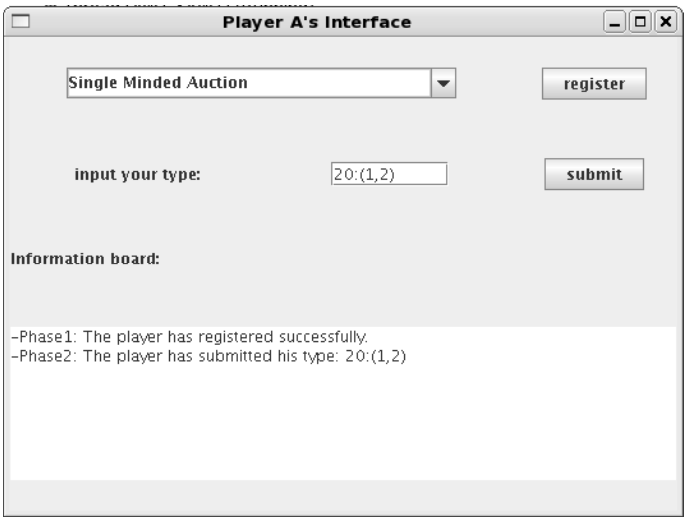

Figure 5: Phase 2: player A

Figure 6: Phase 2: player B

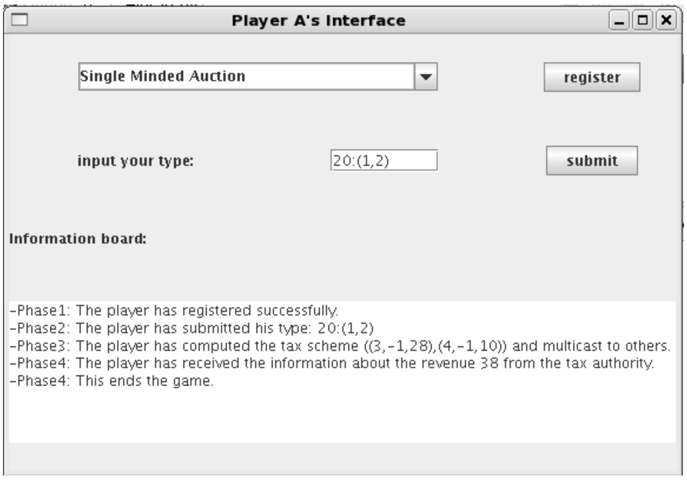

Figure 7: Phases 3 & 4: player A

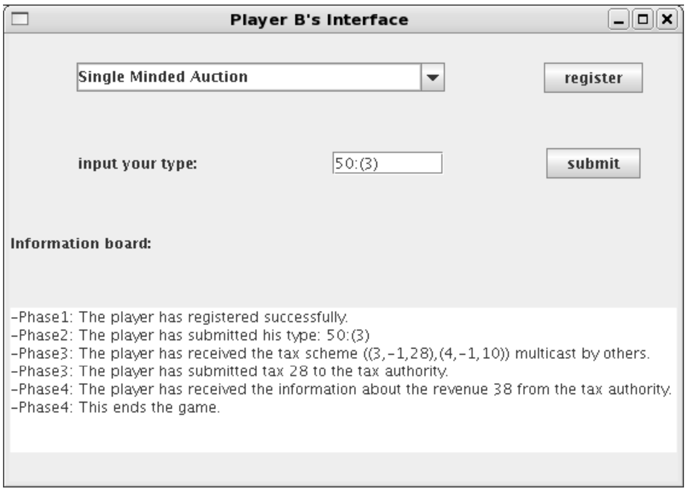

Figure 8: Phases 3 & 4: player B