Real-Time Video Delivery with Market-Based Resource Allocation

Sun Microsystems Laboratories, Inc.
SunConnect, Inc.
Agorics, Inc.

Agorics Technical Report ADd004P

Introduction **1**

1. Overview **3**

2. Components of the demonstration **5**

2.1. Fractal Reserve Banking 7

2.1.1. Interface 9

2.1.2. Usage 9

2.2. The general ongoing auction protocol 11

2.3. Bandwidth-specific auction 17

2.3.1. Modeling the network 17

2.3.2. The bandwidth auctioneer 20

2.4. Bidding agents and their strategies 21

2.5. Deliverators 23

2.6. Application/user interfaces for bidding agents 23

2.7. Interaction of the components 24

3. Bibliography **25**

Introduction

This report documents a real-time video delivery system, developed by Sun Microsystems and Agorics, Inc., to demonstrate how computational resources such as CPU time and ATM network bandwidth may be allocated based on markets, internal to the computer system, on which these resources are bought, sold, and traded. This technique of *agoric computation* (also known as *computational markets*) applies economic insights to the automatic management of computational resources. (For further information on agoric computation, see the attached reports "An Introduction to Agoric Computation" and "Architectural Issues in Agoric Open Systems".) This report provides a detailed technical discussion of the components of the current system and how they interact to produce the behavior demonstrated. These components include:

The term *agoric* derives from the Greek word *agora*, the open square that served as the marketplace and forum for public discussion.

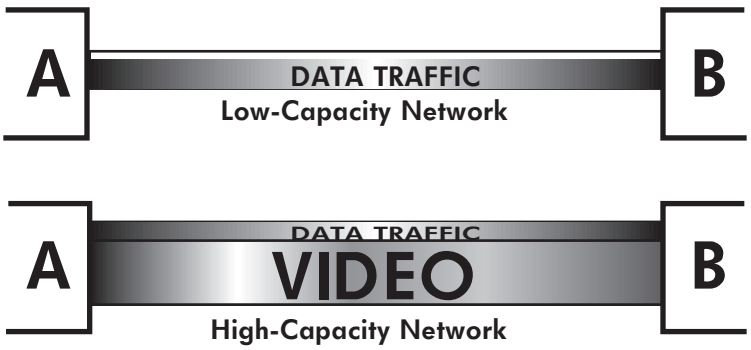
- A system of *banking and currency* to represent a software object's budget and its payments to other objects
- A set of protocols enabling software objects to participate in automated *auctions* of computational resources or composite packages of resources
- *Bidding agents*—software objects that represent application programs in these auctions, expressing the application's resource needs and bidding on behalf of the application to fill them
- *Auctioneers* that each allocate an available resource, in real time, for its highest-valued uses, based on the stated needs of the programs contending for the resource
- *Delivery agents*, such as schedulers and I/O controllers, which provide resources to applications, based on the allocation arrived at by the auctioneer
- *Application and user interfaces* for bidding agents, which translate user preferences or application needs into bidding strategies and state bids for resources to the auctioneers and delivery agents

1. Overview

The *agoric paradox* is the observation that **the cheaper a resource becomes, the more important it is to have automatic management of that resource in a principled fashion.** This is because increased capacity makes it possible to apply that resource to lower-value uses. When a resource is expensive, all uses, in order to be worth the amount they consume, must exceed some minimum value to their users, so all uses of an expensive resource have high and comparable values. Also, because individual uses of an expensive resource have high value, it is both easy and worthwhile to manage the resource manually. When the price per unit of that resource drops by multiple orders of magnitude, uses that have a low value per resource unit become feasible, but these low-value uses can crowd out the high-value uses if there is no way to express the value differences—and manual management of the resource becomes prohibitively difficult, as well as expensive (relative to the value of the use of the resource).

For example, the transition to fiber data communication makes possible bandwidth-intensive applications like video. It might seem, since fiber has so much more bandwidth than previous communications media, that existing network allocation policies (like Ethernet) can simply be scaled. However, in the case of Ethernet, the amount of bandwidth that an application gets is proportional to how much it asks for and how often it asks. In such a regime, network users trying to transmit, for example, electronic mail may find network performance significantly reduced by the presence of video users, even though both the data users and video users would agree that the value per bit of e-mail is far higher than that of video data—a megabyte of capacity would carry an entire day's correspondence for a large company but amount to only a single frame of high-resolution video, which the video user could drop without noticing. Expanding

Fig. 1.1 The Agoric Paradox



Some large companies have had to resort to the outright banning of digital video on internal networks because of the unmanageable impact on network performance for other users.

the overall capacity of the network will not address the problem, because the higher capacity of the network invites still more video use. Non-agoric protocols provide no way to express this value difference. Thus economic abstractions—price and value—*meaningful to the system and automatically taken into account*, are needed in order to manage abundant computing resources. (Such mechanisms must also be flexible enough to accommodate the instances when the video *is* more valuable than other data.)

We do not claim that computational markets are the only possible solution to this problem—only that agoric systems are superior to the status quo, and the most practical solution yet put forward.

2. Components of the demonstration

This report documents a demonstration system that implements techniques for solving the agoric paradox with respect to video. The demonstration shows a video delivery system operating over an asynchronous transfer mode (ATM) wideband network. CPU time and network bandwidth are allocated agorically. Users are provided with a graphical user interface (the *Q-P GUI*) allowing them to specify their preferences with regard to tradeoffs between image quality and price, and displaying the current price being paid for the video delivery circuit.

As video sessions begin and end, the market price of network bandwidth fluctuates in response to overall demand; the picture quality of each user's video changes in response—improving quality due to the availability of more bandwidth when prices fall; graceful degradation of picture quality when the network is busy and bandwidth becomes expensive. The user sets these tradeoff points once, in terms of what each individual level of picture quality is worth to him or her, and can then ignore the settings—the viewer application will automatically implement the user's stated preferences as conditions change.

Fig. 2.1 A sketch of the video viewer. On the right is the video window, displaying the video requested by the user. On the left is a simple quality-price bidding interface, allowing the user to specify how much the user is willing to pay for six different pre-defined levels of video quality.

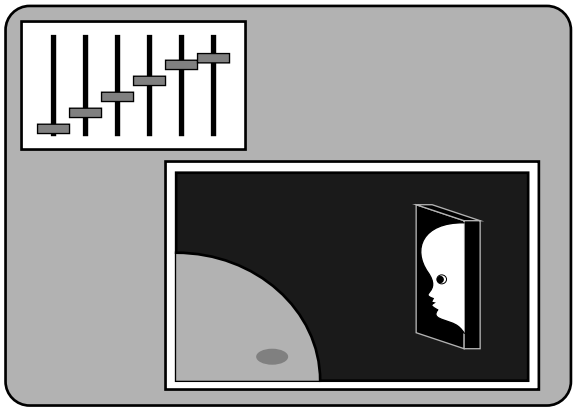
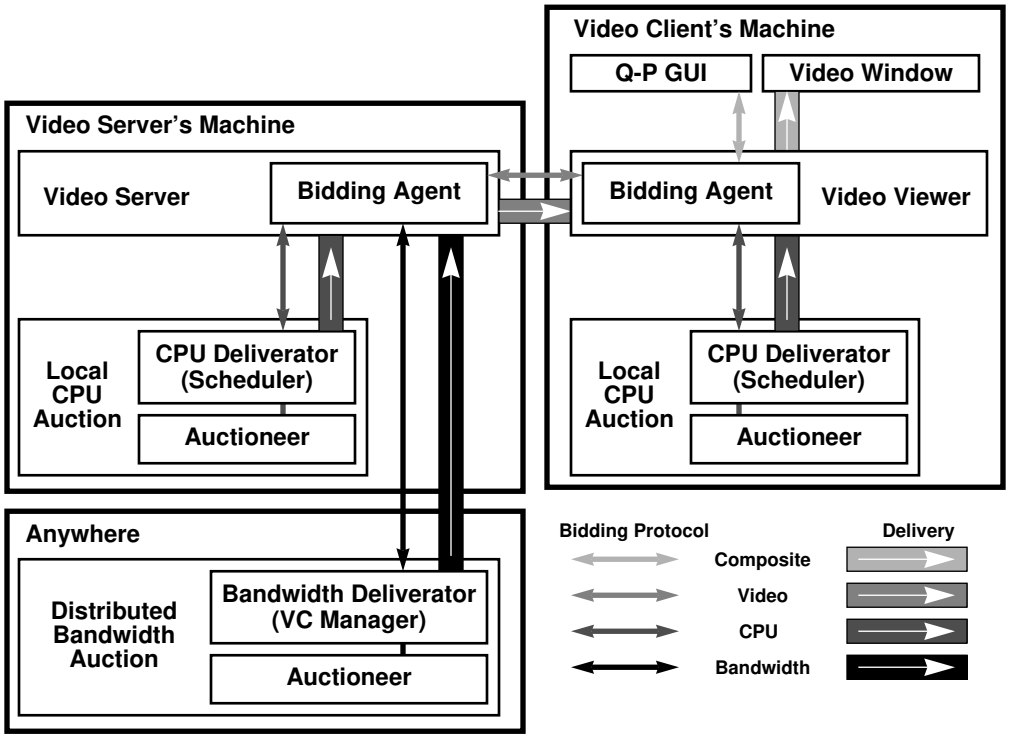


Figure 2.2 shows a schematic diagram of the planned overall structure of the video delivery system. Double-ended arrows indicate the negotiation dialogue between clients and servers, while the wide bars containing single-headed

Fig. 2.2 Overall structure of video delivery system



arrows indicate the delivery of those services to the purchaser. The system consists of:

- **The video viewer.** This client application displays on the user's workstation digital video transmitted over the network from a video server elsewhere on the network. The viewer presents the user with interfaces for selecting the desired video clip and for specifying the maximum prices the user is willing to pay for specific levels of picture quality. The viewer application also includes a bidding agent which translates the user's preferences into a bidding strategy for securing the resources (such as processor time and bandwidth) needed for transmitting and displaying the video
- **The video server.** This application accepts bids from the viewer's bidding agent, decomposes them to bid for the separate resources (processor time and network bandwidth) needed by the server to provide the service, and transmits the video at the agreed-upon level of quality to the viewer.
- **The bandwidth auctioneer** accepts bids from one or more video servers, on behalf of one or more viewer-clients per server, competing to purchase ATM virtual circuits for delivery of video. The bandwidth auctioneer may be on the same machine as any video client or server, or on a separate machine.
- **The bandwidth deliverator** is the virtual circuit manager for the ATM network. It establishes virtual circuits over the ATM network to convey video from servers to clients, as determined by the results of the bandwidth auction. The auctioneer and the deliverator update the allocation and pricing of bandwidth in real time, responding to changes in demand for bandwidth as video sessions begin or end and as users' preferences change.

The present version of the demonstration implements a centralized auctioneer: the auction runs on a single machine of the network, and all clients of the auctioneer must communicate with that machine to participate in the auction. In subsequent implementations, the bandwidth auction will be distributed to enable separation of administration issues and trust-boundary issues, permit a diversity of auction policies to coexist, reduce communications latency, and ensure more effective scaling to larger systems.

We currently implement a separate deliverator for each host in the ATM network because present ATM switch designs make it infeasible to control virtual circuits directly.

- **The CPU auctioneer and scheduler.** Each machine also implements a local auction in processor time. Applications submit bids to a CPU auctioneer, which calculates the results of the auction and tells the CPU deliverator, or scheduler, which processes to run at what times. **Note:** The present incarnation of the demonstration does not implement the CPU auction. The planned design of the system calls for an agoric version of the technique described in "A Scheduling Facility in Support of Multimedia Applications" by Nieh, Northcutt, *et al*.
- **Fractal Reserve Banking.** This is our name for a system of hierarchical accounts representing drawing authority on computational budgets. Payments between software objects take the form of transfer of access to a particular fractal reserve account from the purchaser to the seller. The hierarchical nature of these accounts, as explained in the next section, enables flexibly secure management of budgetary drawing authority among software objects.
- **Interfaces** for communication among these components.

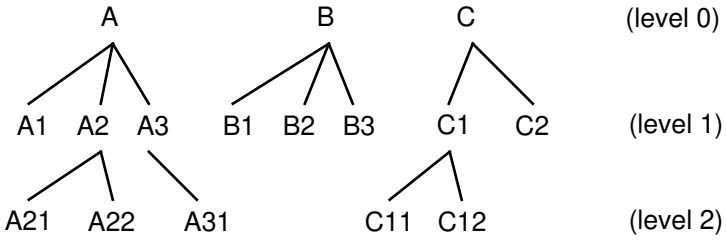
The following sections provide more detailed descriptions of the interfaces to, and current implementations of, the components of the demonstration system.

2.1. Fractal Reserve Banking

The term *fractal reserve banking* refers to a system of accounts that implements hierarchical ownership and drawing authority. The accounts in this system are hierarchical because each can have multiple sub-accounts, each of which is budgeted drawing power on the parent account. The system is "fractal" because it applies the device of fractional reserve banking recursively. The logical relationship of pieces to wholes does not change at different levels of granularity—the system exhibits the fractal property of *self-similarity.*

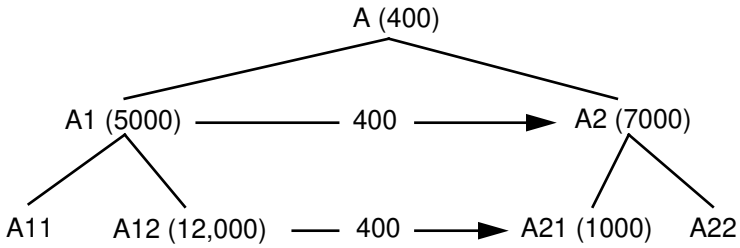
In Figure 2.3, A, B, and C are each the top-level or "root" account of a separate currency. Each top-level account can be thought of as the supply of a single currency. In this model, there is no primitive exchange between currencies; each is completely separate.

Fig. 2.3 Tree of hierarchical accounts



A hierarchical account can create sub-accounts with arbitrary budgets. The budgets an account may assign to its subaccounts are unlimited. When a subaccount within that account needs to transfer funds outside of the ancestor account, however, the amount is limited by the budget of the ancestor account. This is because the budgets of their respective ancestor accounts must be balanced as well. In Figure 2.4, any amount (up to the budget of A11) can be transferred from A11 to A12, because these are totally internal to the A1 par-

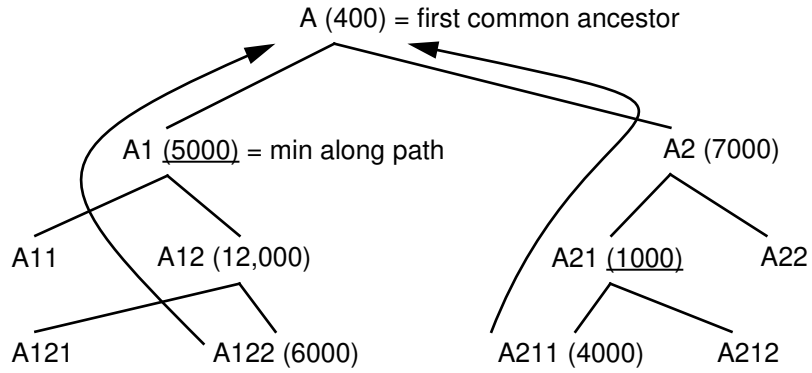
Fig. 2.4 Transfer of funds. All accounts along the path from the paying account to the common ancestor are decremented by the amount of the transfer (in this case, 400 units); all accounts that are ancestors of the payee account are incremented by that amount.



ent account; however, the transfer of 400 credits from A12 to A21 must be covered by a corresponding transfer from A12's parent A1 to A21's parent A2. The maximum amount for such a move is A1's budget of 5,000 tokens.

In general, the amount that can be transferred from one account to another anywhere in the hierarchy is the minimum of the budgets of the accounts on the path from the donor account to the nearest ancestor it has in common with the recipient account (not including the common ancestor account itself).

Fig. 2.5 Nearest common ancestor for two accounts



For example, in Figure 2.5, the most that could be transferred from A122 to A2 or any of its descendants is 5,000 tokens, the minimum among the budgets of A122 and its ancestors A12 and A1. The most that could be transferred from A211 to A1 or any of its subaccounts is 1,000, the minimum of the budgets of A211, A21, and A2.

The motivation for fractal reserve banking is to enable a server (any software object that provides computational services to other objects) to more easily subcontract for what it needs in order to provide service to its clients. The client can, in effect, write a blank check of limited size—"I will pay you up to but not more than $X for this specified amount of service."

The server can then, in turn, use subaccounts of this account to pay its suppliers, budgeting fractions of that amount for each of the more basic services it needs (CPU, bandwidth). The fractal nature of the accounts allows the total of these budgets (but not the total of expenditures) to exceed the total contents of the account.

2.1.1. Interface

Interfaces to system components are presented in this report in CORBA/IDL (the Interface Description Language of the Common Object Request Broker Architecture), the proposed standard language for the definition of interobject communication interfaces. Fragments of test code are in C++, the actual implementation language for the components of the demonstration.

The public messages to which valid implementations of Account will respond are withdraw, budget, deposit, balance, and isSameCurrency. New currencies are created with the static method newCurrency.

From **frbank.idl**:

```
interface Account {

   //...

   Account withdraw (in Integer amount)
    raises (InsufficientFunds, NegativeAmount, WithdrawFromRoot);

   Account budget (in Integer amount)
    raises (NegativeAmount);

   Integer deposit (in Account source, in Integer amount)
    raises (InsufficientFunds, NegativeAmount, DifferentCurrency);

   Integer balance (in Account destination, in Integer max)
    raises (NegativeAmount, DifferentCurrency);

   boolean isSameCurrency (in Account other);

   static Account newCurrency ();
}
```

The withdraw message instructs the account to create a sibling account and transfer amount from its own balance to the new account. The result returned is a pointer to the new account. Because this new account is created by its sibling, its budget must be deducted from the budget of the original account; money is conserved between siblings.

The budget message instructs the account to create a new subaccount, with an initial budget of amount, which (since it is simply drawing authority on the parent account) can be arbitrary. It returns a pointer to the new subaccount.

The deposit message transfers amount from the account source to this account.

The balance message takes as arguments an amount max and another account destination; it addresses the question "Could this account transfer max tokens into destination?" It returns the minimum of max and the greatest allowable transfer (equal to the minimum of all the balances of ancestors from this account's parent to the common ancestor of this and destination). The candidate amount max is present to avoid infinities in the protocol.

The static method newCurrency is what we call a "pseudo-constructor". It is used to create a new currency.

2.1.2. Usage

The suffix "Impl" (short for implementation) is a CORBA convention indicating that "FooImpl" is a specific implementation of a software object satisfying the interface "Foo". A C++ regression test file called frbank-t.cc illustrates the use of our AccountImpl:

From **frbank-t.cc**:

```
#include “frbank.hh”
int main (int ac, char *av[])
{
   //...

   AccountRef root = Account::newCurrency ();
   AccountRef foo = root->budget (100);
   AccountRef sub = foo->budget (1000);
   AccountRef bar = root->budget (200);
```

First, a new currency is created, with root as its top-level account. Subaccounts foo and bar are created, with initial budgets of 100 and 200 tokens, respectively, as well as a subaccount of foo called sub with a budget of 1000.

From **frbank-t.cc**:

```
   cerr << "foo: " << foo << "\n";
   cerr << "sub: " << sub << "\n";
   cerr << "bar: " << bar << "\n\n";
```

At this point, the three accounts, when printed with the << operator, look like this:

From **frbank-t.reg**:

The filename extension “**.reg**” indicates the output from a regression test.

```
foo: AccountImpl(level: 1, budget: 100)
sub: AccountImpl(level: 2, budget: 1000)
bar: AccountImpl(level: 1, budget: 200)
```

Next, 20 tokens are transferred from sub to bar:

From **frbank-t.cc**:

```
bar->deposit (sub, 20);
   cerr << "foo: " << foo << "\n";
   cerr << "sub: " << sub << "\n";
   cerr << "bar: " << bar << "\n";
```

Printing the three accounts reveals the changed balances:

From **frbank-t.reg**:

```
foo: AccountImpl(level: 1, budget: 80)
sub: AccountImpl(level: 2, budget: 980)
bar: AccountImpl(level: 1, budget: 220)
```

The budget of sub’s parent foo has also been diminished by 20 tokens (as in Figure 2.4 above). The amount that can be transferred between two accounts is limited by the budgets of the donor account’s ancestors, as the rest of frbank-t.cc shows.

From **frbank-t.cc**:

```
   try {
     bar->deposit (sub, 200);

   } catch (UserException * ex) {
     cerr << "caught " << UserExceptionRef(ex) << "\n";
   }
```

The amount of the transfer is greater than the budget of foo, so it raises the InsufficientFunds exception:

From **frbank-t.reg**:

```
caught InsufficientFunds(shortfall: 120, whereNeeded:
                         AccountImpl(level: 2, budget: 980))
```

2.2. The general ongoing auction protocol

The file **auction.idl** describes the interfaces for the generic Auction protocol. The major players in the generalized Auction are:

- the Deliverator, which provides some resource or service and charges for its use;
- the Auctioneer, which accepts bids from applications that wish to use the resource, and which decides the allocation and salePrice for the resource; and
- various Bidding Agents which submit bids to the Auctioneer to buy service for themselves or other programs.

The Bidder shown here might be an application program buying for itself. However, it might instead be a server program acting on behalf of a client to which it is providing some service, accepting bids from the client in very different terms than those understood by the Auctioneer, and translating them into the Auctioneer's terms in order to buy the resources to provide that service. A video-viewer client might express its preferences in terms of specific levels of quality of service (monochrome vs. color, frame rate, resolution). The server providing the video may need a variety of resources (ATM bandwidth, CPU time, disk bandwidth) to deliver service, and have to bid for each resource in a separate auction, in the terms specific to that resource. Section 2.4 presents the set of interfaces for bidding agents. For discussion of the general Auction protocol, assume a simple Bidder purchasing a resource for itself.

Fig. 2.6 Message-passing sequence to establish communications for bidding

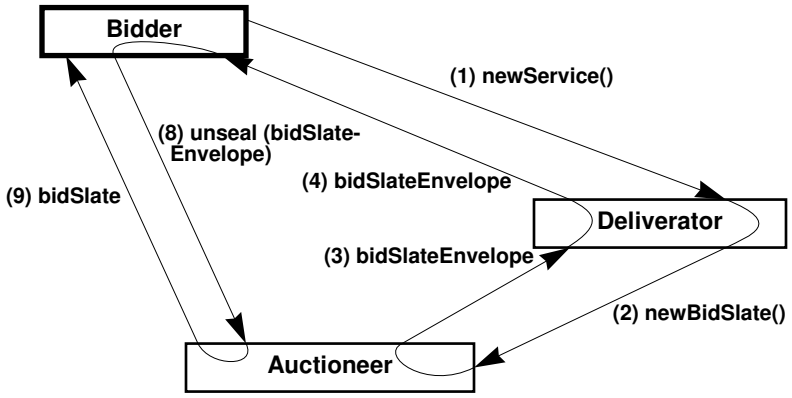


Figure 2.6 shows the sequence of message-passing that occurs when the protocol is initiated by the Bidder, which sends the newService message to a Deliverator. The complexity of this protocol results from the necessity of making the auctioneer distinct from the seller (the Deliverator). A seller running its own auction could, in a properly-encapsulated system, undetectably defraud its clients; the auctioneer therefore needs to be a "third party" indepen-

dent of both sellers and buyers. In addition, the Deliverator need believe only the Auctioneer about what quantity should be supplied and what price charged.

```
interface Deliverator {

   BidSlateEnvelope newService (ServiceDesc desc,
                                Account expenses);
};
```

The newService message specifies what the bidder wishes to buy (via the ServiceDesc argument) and where the funds to buy the service are to come from (the Account object expenses). ServiceDesc is not defined by this protocol because it will vary with what is being auctioned; the file **auction.idl** describes only the abstract protocol for this kind of market. A market in a particular commodity will define a subclass of ServiceDesc understood by all participants in that market. For example, the market in distributed bandwidth defines a derived CircuitDesc, our implementation of which is described in Section 2.3.1.

The newService message returns a BidSlateEnvelope object, which the Deliverator gets by sending the newBidSlate message to the Auctioneer:

```
interface Auctioneer {

   BidSlateEnvelope newBidSlate (ServiceReactor reactor,
                                 out Evictor evictor);

   BidSlate unseal (BidSlateEnvelope envelope)
      raises (AlreadyOpened, NotSealedByMe);
};
```

The newBidSlate message takes as its arguments a ServiceReactor object (newly created by Deliverator according to the ServiceDesc parameter it got from the bidder) and the right to assign the value of a variable giving Deliverator access to an Evictor object created by the Auctioneer. (The C++ function call for this looks like myAuctioneer–>newBidSlate(service, &evictor).) The purpose of Evictor is to enable the Deliverator to withdraw service from any service consumer (the bidder or its client application) that is in arrears. In so doing, it removes that consumer from the auction. Evictor accepts the message evict:

```
interface Evictor {

   void evict ();
};
```

ServiceReactor objects contain the terms for this particular sale: some quantity of whatever commodity the Deliverator sells, and a sale price for that amount of service. The Auctioneer sets these to initial values of zero until the Bidder later places bids for some quantity of the service.

```
interface ServiceReactor : ServiceTicker {

   oneway void notice (Integer newQuantity, Integer newSalePrice);

   oneway void evictionNotice ();

   ServiceTicker view ();
```

```
    /*
     * One that just stores state and notifies reactors
     */
    static ServiceReactor make (ServiceDesc desc);
};
```

Since the quantity and price attributes inside the ServiceReactor always change at the same time, a single notice message is used to specify their new values. (ServiceReactor can therefore be considered an atomic reactor.)

In general, View objects (such as ServiceTicker) and their corresponding Reactors are complementary: The Reactors are subclasses of the corresponding View class, but another object can post a Reactor on a particular View to accept notifications of status changes.

ServiceTicker is the superclass of ServiceReactor; whenever a ServiceReactor receives the view request, it creates and returns a ServiceTicker object that provides readonly access to the ServiceReactor's attributes: the ServiceDesc being bid upon, the (initially zero) quantity and price, and whether or not the ServiceReactor is still participating in the auction (the isEvicted bit).

```
interface ServiceTicker {

    readonly attribute ServiceDesc serviceDesc;

    readonly attribute Integer quantity;

    readonly attribute Integer salePrice;

    readonly attribute boolean isEvicted;

    void addReactor (ServiceReactor reactor);
};
```

The addReactor message is used to post a "watchdog" object called a reactor on the ServiceTicker receiving the message. Such reactors (a subclass of ServiceReactor) will receive a notification whenever quantity or salePrice changes, instead of having to poll the ServiceTicker.

Auctioneer returns a BidSlateEnvelope to the Deliverator; this is simply an opaque object for secure delivery, by the Deliverator, of another object called BidSlate from the Auctioneer to the bidder without the Deliverator being able to see inside it.

```
interface BidSlateEnvelope {
};
```

Once the bidder receives the BidSlateEnvelope, it can ask the Auctioneer to unseal it, which returns the enclosed BidSlate. As mentioned above, this security is to guarantee the independence of the Auctioneer from the seller. A BidSlateEnvelope can be unsealed only once; subsequent attempts to unseal it will signal an exception. Thus a bidder that successfully unseals a BidSlateEnvelope knows that it is starting with exclusive access to the BidSlate inside.

IDL has both readonly and non-readonly attributes. Declaring a non-readonly attribute in an interface implies two distinct messages that will be accepted by objects conforming to that interface: one to set the attribute and one to read its current value. Both are general-purpose messages that can invoke other behavior as well.

```
interface BidSlate {

    attribute sequence<BidSegment> bidSegments;
```

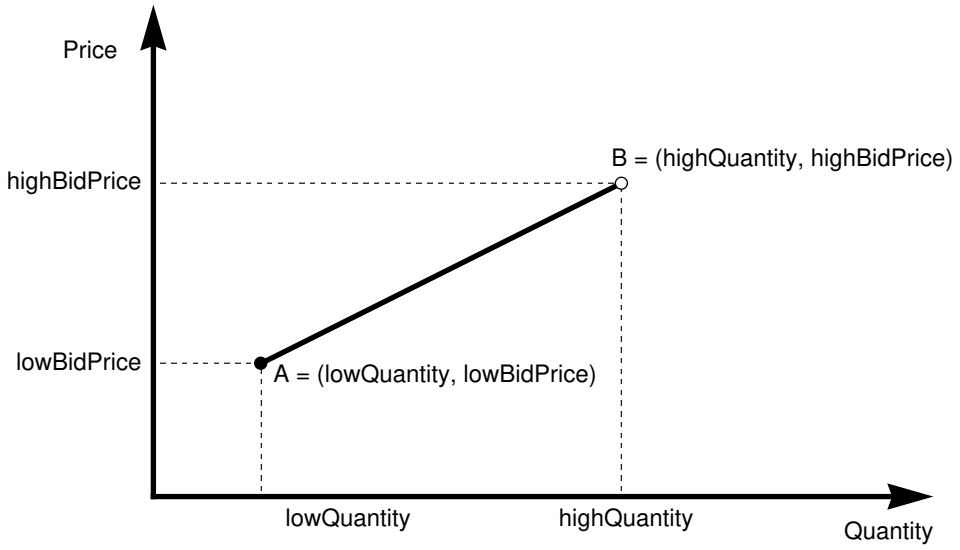
```
    readonly attribute ServiceTicker view;
};
```

The BidSlate object is the bidder's sole interface to the Auctioneer; in it, the bidder specifies the piecewise-linear function describing what price the bidder is willing to pay as a function of quantity of service received. This function is represented as a set of BidSegments.

```
struct BidSegment {
    Integer lowBidPrice, lowQuantity;

    Integer highBidPrice, highQuantity;

    static BidSegment make (Integer bidPrice, Integer quantity);
};
```

Each BidSegment describes a straight-line segment on a graph of price as a function of quantity. It actually represents an infinite number of bids (ordered

Fig. 2.7 One BidSegment (infinitely many bids)

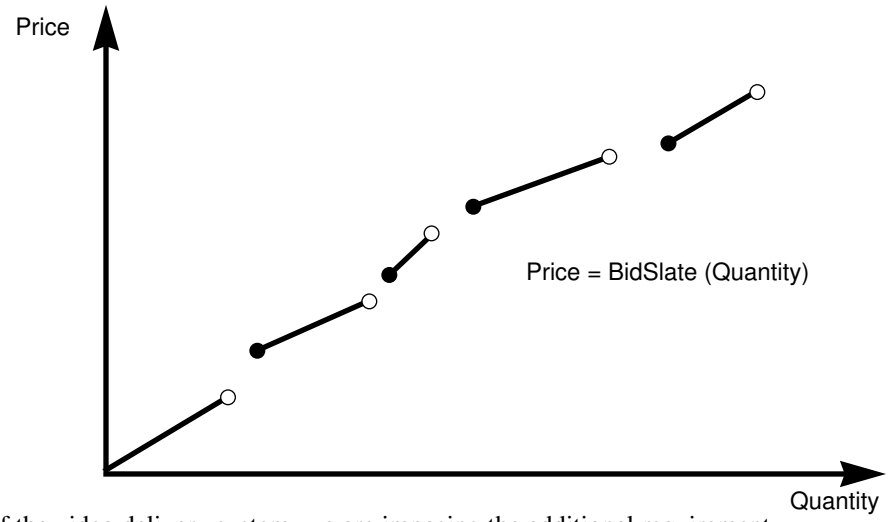


pairs of price and quantity values), one for any particular point along the line segment described. Bids are mutually exclusive; only one will be taken. (The make message implements the special case of a BidSegment consisting of a single point.) These intermediate bids are linearly interpolated from the values at the endpoints. We explicitly assume that a bid of (Q,P) means "I am willing to buy at least Q, at no greater cost than P."

A BidSlate consists of a set of BidSegments that together define the bidder's entire price-quantity function (Figure 2.8). Each BidSegment is a linear piece of the piecewise-linear function described by the BidSlate.

The BidSegments of a valid BidSlate must be mutually exclusive; the same price or quantity can not be in the range of more than one BidSegment. The set of BidSegments should be sorted in increasing order of both price and quantity so that the function described by BidSlate is monotonically increasing. To keep the algorithms simple and uniform, it is further required that the BidSlate start with a BidSegment whose lowBidPrice and lowQuantity are both zero. However, a BidSlate may be discontinuous. (In the current incarna-

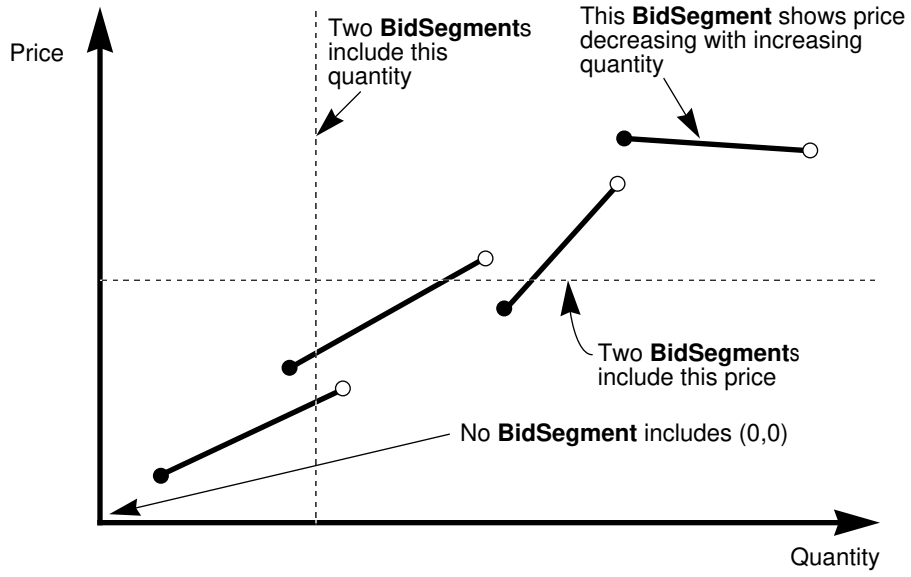
Fig. 2.8 A BidSlate is composed of arbitrarily many BidSegments



tion of the video delivery system, we are imposing the additional requirement that every BidSegment be a single point; i.e., both prices and quantities are quantized.)

Figure 2.8 displays a BidSlate which has several things wrong with it.

Fig. 2.9 An invalid BidSlate



The requirement that a BidSlate include (0,0) as a bid simplifies the Auction protocol by ensuring, in combination with the requirement of monotonic increase, that every BidSlate can be satisfied (if need be, by the delivery of zero quantity for zero price). A bid to acquire some quantity of channel capacity for zero dollars is unreasonable (unlikely to be fulfilled if other bidders are in the auction).

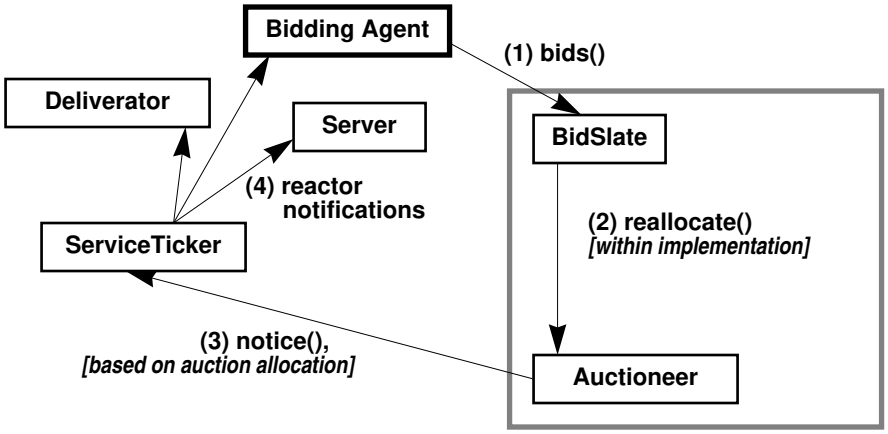
At this point, the bidder has sent a newService request to the Deliverator and received a BidSlate object as a result. The BidSlate is initially empty; the bidder creates and changes its bids by setting the bidSegments attribute (containing the actual ordered sequence of BidSegments). The BidSlate object is created and implemented by the Auctioneer because the Auctioneer

must react whenever the bidder changes the BidSlate's contents. In addition, the Deliverator only has to believe the Auctioneer about what quantity should be supplied and what price charged.

The Auctioneer takes in bids via BidSlates from multiple competing bidders. The prices specified by each BidSlate represent the value to that application of having a particular quantity of the commodity available to it. From these statements of value, the Auctioneer computes an allocation of the available supply of that resource that awards the resource to the highest-valued uses among the set of multiple bidding clients, based on the assumption that the stated bidPrices accurately reflect value.

Based on that result, the Auctioneer changes the values of salePrice and quantity visible through the ServiceTickers corresponding to all of the bidders in the auction. All reactors posted on an individual ServiceReactor and its ServiceTickers, including Deliverator and the bidder, are notified of the assigned values and can react to them.

Fig. 2.10 Events of bidding procedure



Once the Auctioneer sets the quantity to be delivered, the Deliverator reacts by delivering to the client the quantity of service allocated to it. For example, in a simple bandwidth auction, quantity is the bandwidth to be allocated to the client's virtual circuit. The Auctioneer may change that at any time, and the Deliverator reacts by allocating a different share of bandwidth to that virtual circuit and informing the application (via a ServiceTicker and reactor notification) so it can adjust.

The attribute salePrice allows the Deliverator to know how much it should charge, and the application how much it should pay. The amount the application must pay the Deliverator need not be the same as salePrice, but salePrice is a candidate amount set by a mutually-trusted third-party Auctioneer. Any financial arrangement between the Deliverator and the application which is mutually acceptable (such as salePrice plus some markup) is fine.

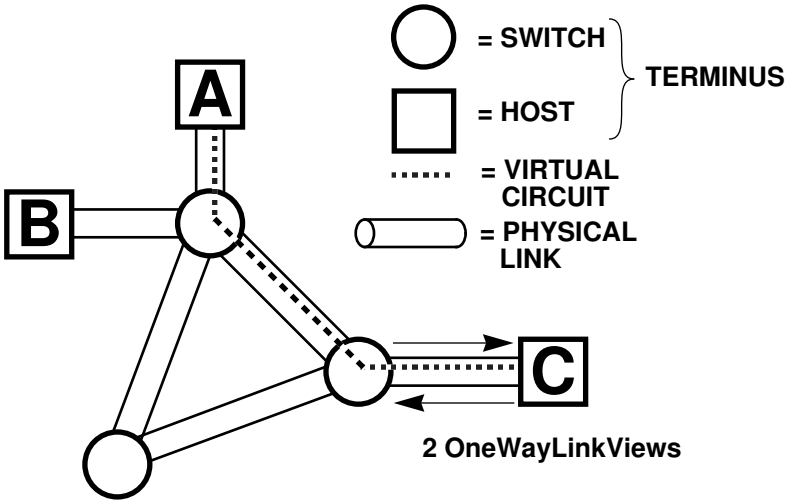
If the application doesn't pay the Deliverator enough to satisfy it, the Deliverator can evict the application from the BidSlate, by sending the evict message to the Evictor specified in the original newBidSlate request to the Auctioneer and removing that bidSlate from the auction.

2.3. Bandwidth-specific auction

2.3.1. Modeling the network

Figure 2.11 introduces a schematic representation of an ATM network, with squares representing hosts, circles representing ATM switches, and dotted lines representing virtual circuits between specific hosts. The file **netmodel.idl**

Fig. 2.11 Representation of an ATM network



contains the interface descriptions for the various components of such a representation.

Hosts and switches are represented by Terminus objects; each physical fiber linking two termini is represented by a pair of OneWayLinkViews, one for each direction. Each terminus has one set each of incoming and outgoing phys-

```
interface Terminus {

   enum TType { HOST, SWITCH, DISPOSED };

   readonly attribute TType tType;

   readonly attribute sequence<OneWayLinkView> outLinks;

   readonly attribute sequence<OneWayLinkView> inLinks;

   readonly attribute string name;

   readonly attribute Integer index; // in network model's termini
};
```

ical fibers; these are represented by the two sequences of OneWayLinkViews called inLinks and outLinks. The attribute tType indicates whether the terminus is a host or a switch, or is out of service (DISPOSED). Finally, each Terminus is identified by both a name and an index number; the latter, as we shall see later, identifies this Terminus to a NetworkView object.

The location and direction of a OneWayLinkView object are specified by the two termini it joins, called from and to. The link knows its positions in the inlinks or outlinks sequence of each Terminus (fromIndex and toIndex). Its

attributes bandwidth and price represent those characteristics of the real fiber it represents.

```
interface OneWayLinkView {

  readonly attribute Integer bandwidth;
  readonly attribute Integer price;

  /* NIL only if disconnected */
  readonly attribute Terminus NIL_OK from;
  readonly attribute Terminus NIL_OK to;

  readonly attribute Integer fromIndex; // in from's outLinks table
  readonly attribute Integer toIndex; // in to's inLinks table

  void addReactor (OneWayLinkReactor reactor);
};
```

A OneWayLinkView object accepts the addReactor message whose argument reactor is of the class OneWayLinkReactor or a subclass of it. Subsequently, whenever one of the attributes of the OneWayLinkView changes, it will send to reactor one of the notice messages defined for OneWayLinkReactor. In particular, the noticeDisconnect message tells the

```
interface OneWayLinkReactor : OneWayLinkView {

  oneway void noticeBandwidth (Integer bandwidth);

  oneway void noticePrice (Integer price);

  oneway void noticeDisconnect ();

  OneWayLinkView view ();
};
```

In general, the **view** message lets a **Reactor** pass along the capability to read the attributes of the **View** without passing the ability to set them.

reactor when the OneWayLinkView has become disconnected—that is, when its from and to attribute have become NIL. The view message, sent to the reactor by another object, returns the OneWayLinkView that it is watching.

Similarly, a NetworkView object is watched by NetworkReactor objects set by an addReactor message (each FooReactor class is a subclass of the corresponding FooView). The NetworkView represents the network as a sequence of termini. It notifies its reactors of changes in its state with another

```
interface NetworkView {

  readonly attribute sequence<Terminus> termini;

  void addReactor (NetworkReactor reactor);
};
```

set of notice messages: noticeTerminus when a new Terminus has been added to the network; noticeDisposal when a Terminus is withdrawn from service; noticeConnection when a new OneWayLinkView has been created from one Terminus to another; and noticeDisconnection when a OneWayLinkView goes out of service.

Another subclass of NetworkView called NetworkEditor is used to edit a network description to keep it in conformity with the actual physical network.

```
interface NetworkReactor : NetworkView {

    oneway void noticeTerminus (Terminus terminus);

    oneway void noticeDisposal (Terminus terminus);

    oneway void noticeConnection (OneWayLinkView link);

    oneway void noticeDisconnection (OneWayLinkView link);

    NetworkView view ();

    static NetworkReactor make ();
};
```

Messages are provided to establish or disable hosts, switches, and OneWayLinkViews; set the attributes of OneWayLinkViews within the model; and return the NetworkView on which this NetworkEditor operates.

```
interface NetworkEditor : NetworkView {

    Terminus newSwitch (string name);

    Terminus newHost (string name);

    void dispose (Terminus terminus);

    OneWayLinkView connect (Terminus from, Terminus to);

    void disconnect (OneWayLinkView link);

    void setBandwidth (OneWayLinkView link, Integer bandwidth);

    void setPrice (OneWayLinkView link, Integer price);

    NetworkView view ();

    static NetworkEditor make ();
};
```

Modeling the network is important because ATM networks, in their present form, do not have flow control. An analogy can be made to the difference between plumbing using pipes and plumbing using aqueducts. If you have a tank of water at the top of a hill and wish to send water downhill through a system of aqueducts, you need to come up with a model of the aqueduct system and use the model to calculate how much water you can send safely down the network. If more water is sent downhill than the aqueduct can handle, it simply overflows, spilling some of the water. In a network, this means dropped packets and lost data.

If you instead have a network of pipes, you can simply open the valve, and the pipes will admit only what they can deliver. Pipes provide feedback, in the form of back-pressure, to the tank, controlling the amount of water that flows downhill. Aqueducts provide no feedback from the real object, so an accurate model is necessary to provide feedback. This is why we are throttling the network at the Deliverator on each host—making sure that only as much data as can be accommodated is sent down the ATM channel—and why it's necessary to have an accurate model of the ATM network.

2.3.2. The bandwidth auctioneer

The file **netauction.idl** connects the **netmodel**, **auction**, and **bidder** interfaces to implement an auction specific to the negotiated sale of network bandwidth. The ServiceDesc for the network auction is called CircuitDesc. It contains a description of a virtual circuit in terms of a sequence path of OneWayLinkViews, as well as the beginning and end of the path in terms of position in the outCircuits and inCircuits tables of the starting and ending hosts.

```
interface CircuitDesc : ServiceDesc {

   readonly attribute sequence<OneWayLinkView> path;

   readonly attribute Integer fromIndex; //in starting Host's outCircuits

   readonly attribute Integer toIndex; //in ending Host's inCircuits
};
```

The Auctioneer subclass which understands CircuitDescs is called NetAuctioneer. It implements a centralized auction in distributed bandwidth. The netAuctioneer is supplied with a NetworkEditor so it can set the prices of the individual OneWayLinkViews in the network via the setPrice message. It does not otherwise edit the network model (as if it had been given only a NetworkView).

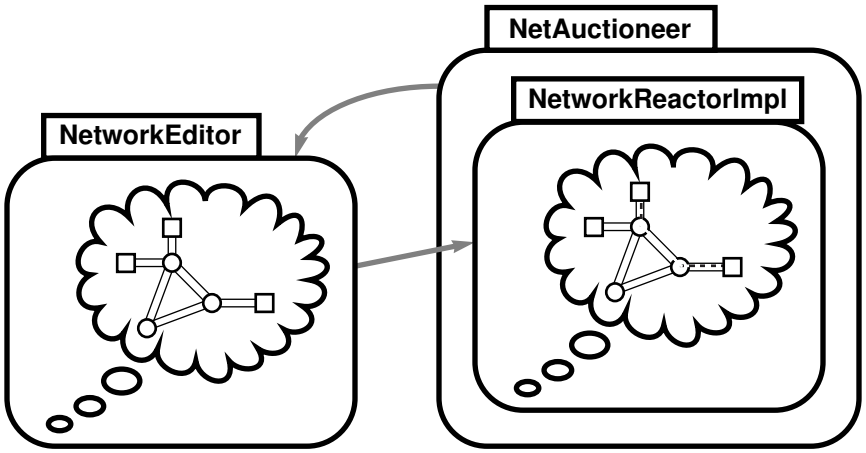
```
interface NetAuctioneer : Auctioneer {

   readonly attribute NetworkView network;

   CircuitDesc connect (sequence <OneWayLinkView> path);

   CircuitDesc autoConnect (Terminus from, Terminus to,
                  Integer expectedBandwidth);

   static NetAuctioneer make (NetworkEditor net);
};
```

NetAuctioneer responds to changes in the network model as well as to the coming and going of circuits and the placing of bids via BidSlates. If the network model changes in such a way that live virtual circuits are disconnected, the users of those circuits are evicted.

NetAuctioneer also takes the autoConnect request, by which the requester asks NetAuctioneer to figure out a path between the hosts from and to. It may try to find the path of least cost that satisfies the expected need for bandwidth stated by the requester. This message provides the hook for adding adaptive routing in the future.

Both NetAuctioneer and NetworkEditor have an associated NetworkView, a model of the same real network. NetworkEditor makes changes in the model in response to both real-world news of termini or links going out of service, and in response to NetAuctioneer setting the prices of available links. Implementations of NetAuctioneer will contain a NetworkReactor which will respond to notifications of changes in the network so that the NetAuctioneer can update the prices and allocation of the network accordingly.

Fig. 2.12 Mutual dependence for network models



The current NetAuctioneer computes an overall allocation of bandwidth to circuits which maximizes total aggregate value delivered to the community of users, assuming that their bid prices are an accurate statement of the value to them of various quantities of bandwidth. It then charges them an amount, the *opportunity cost*, that we believe causes it to be in their interest to "tell the truth."

The current implementation of the NetAuctioneer is unrealistic in two regards. First, it is a single centralized entity which requires a global model of the network, rather than a distributed network of auctioneers each of whom have local knowledge only of parts of the net. Second, it computes both allocations and prices by combinatorial search with exponential cost, rather than approximating ideal results in exchange for a reasonable computational burden. This latter tradeoff is itself a ripe opportunity for the application of agoric principles.

2.4. Bidding agents and their strategies

This section defines a number of interfaces for communication between buyers and sellers. As mentioned in Section 2.2, the end-user of a purchased service may express needs in terms very different from those understood by the end-suppliers. In the example of client-server video, the bidding agent at the viewing end may express levels of quality of service in terms of composite quantities—frame rate, resolution, color depth—rather than the separate resources—network bandwidth, CPU time—which the server must purchase to provide the service. The various programs along the way each negotiate for the resources they need, using the protocols and descriptions of service that their suppliers understand.

Figure 2.2 shows the bidding agents of several programs interacting to provide service to a user. The video viewer separates the user's request into the resources it needs to provide the requested service—CPU on the local machine, and video from the video server. The video server in turn requests from its suppliers the resources it needs—CPU time, and an ATM virtual circuit to the client. This decomposition by an intermediate agent of a requested service into the resources needed to fulfill the request is the equivalent, in the performance domain, of the subcontracting that goes on, via object-oriented programming, in the domain of program correctness.

The file **bidder.idl** describes general interfaces for auction-specific bidding protocols and for bidding agents that employ them. The QualityOfService interface is defined without any particular methods for the sake of generality:

```
interface QualityOfService {

   readonly attribute string name;
};
```

The other interfaces are subclasses of QualityOfService. SingleQOS specifies a single discrete service, with a single indifference price telling the bidding agent how badly you want that service. If the service is available at the stated price or lower, it is provided; if not, nothing is provided. "Nothing" is always available; in fact, the delivery of nothing (NoService) is always listed as an available quality of service, so that denial of service can be stated by the bidding agent as delivery of NoService—a kind of delivery, rather than a special case. (If NoService is delivered, an honest bidding agent will ensure that the bidder's Account is not charged.)

```
interface SingleQOS : QualityOfService {

   attribute Integer indifferencePrice;
};

interface NoService : QualityOfService {
};
```

The interface BiddingAgent is obtained by a client in response to a request for service made to a server. (The request itself is service-specific, so no general protocol is given as part of BiddingAgent.) It is presumed that the request takes as a parameter an Account object via which the client pays the server.

```
interface BiddingAgent {

   sequence<QualityOfService> qualities ();

   readonly attribute Integer deliveredQuality;

   readonly attribute Integer salePrice;

   void addReactor (QOSReactor reactor);
};
```

The client then inquires of the BiddingAgent object what qualities of services are available (expressed in terms of subclasses of QualityOfService), and assigns indifference prices to express the degree to which each is desired. An honest bidding agent will try to bid in such a way as to get the client the most for its money, according to the stated preferences, and to arrange for the client's Account to be charged accordingly. The BiddingAgent informs the client what was obtained and for how much via the deliveredQuality and salePrice attributes.

Like some of the objects described in the previous section, BiddingAgent also has a reactor subclass, QOSReactor, subclasses of which can be set to receive notification of any changes in the BiddingAgent's status and act accordingly.

```
interface QOSReactor : BiddingAgent {

   oneway void notice (Integer quality, Integer price);

   BiddingAgent view ();
};
```

2.5. Deliverators

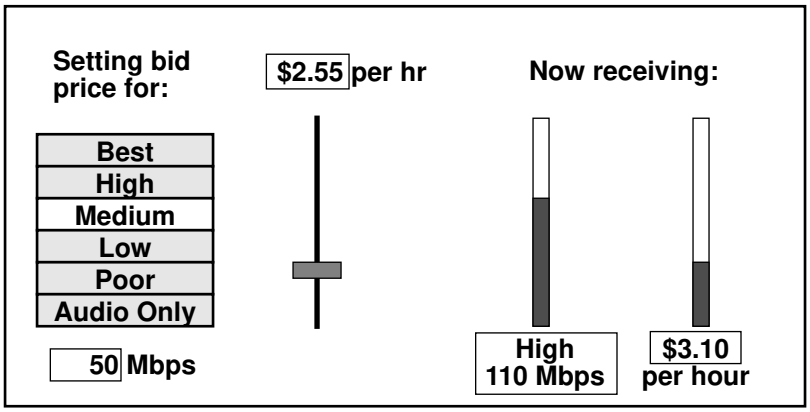
The Sun Microsystems ATM host interface architecture for managing ATM networks is particularly well-suited to implement agoric management of bandwidth. This interface treats a physical ATM link as a separable bundle of 256 bandwidth groups, subsets of which can be assigned to the initial (outgoing) segment of an ATM virtual circuit. This makes the individual physical links of an ATM network nicely divisible for assignment to particular virtual circuits in response to the result of the bandwidth auction.

2.6. Application/user interfaces for bidding agents

The user interface for the video client's bidding agent allows the user to express the maximum price he is willing to pay for each of several levels of video quality, where "video quality" is defined as a set of values for specific components like resolution, bit depth, frame rate and so forth. (These price-quality pairs become single-point BidSegments that together make up a BidSlate (see Figure 2.8).

The current user interface to the bidding agent (Figure 2.13) defines a set of

Fig. 2.13 The Video Palette



The user interface presented here is one example. The design of the QP agent user interface will evolve as the development of the system progresses.

predefined and named quality levels and provides sliders to set, for each quality level, the maximum price the user is willing to pay for that level of video quality. It also displays the current level of quality the user is receiving and the price paid for it.

Often, the system's default bidding strategies will be adequate, and the user won't need to deal with the bidding agent interface at all.

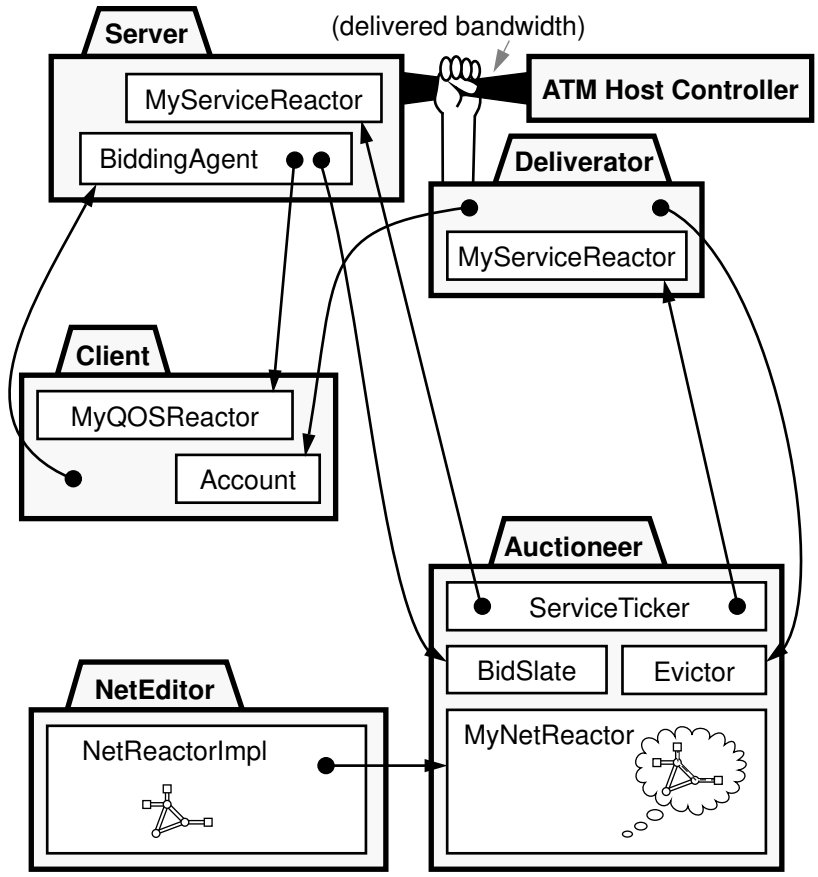
This design shows that simple interfaces can capture a user's price-vs.-quality preferences in a straightforward and uncomplicated manner, even when the criteria which define a particular quality level are complicated and multiple. The

user interfaces to bidding agents for different applications will be specific to that application, but can incorporate some of the same simplifying design concepts shown here.

2.7. Interaction of the components

Figure 3.1 shows a broad view of how the various objects which make up the system exchange messages to implement the intended behavior. The black dots represent proxies for objects at the other end of the arrows.

Fig. 3.1 Message-passing between objects



Reactors are used for one process to monitor another. Process-watching is implemented by notification, not by polling; when an object of type Foo undergoes a change which other processes should know about, it transmits a notification to every process that has posted a FooReactor on it.

The architecture of the system is secure when implemented on top of a secure communications medium.

3. Bibliography

[1] Anderson, M., Pose, R. D., and Wallace, C. S., "A Password-Capability System", *The Computer Journal*, Vol. 29, No 1, 1986.

[2] Axelrod, Robert, *The Evolution of Cooperation.* New York: Basic Books, 1984.

[3] Coase, R. H., "The Nature of the Firm," in *Economica: New Series* (1937), Vol. IV, reprinted in Stigler, G. J., and Boulding, K. E. (eds.), *Readings in Price Theory.* Chicago: Richard D. Irwin, Inc., 1952.

[4] Cocchi, Ron; Shenker, Scott; Estrin, Deborah; and Zhang, Lixia, "Pricing in Computer Networks: Motivation, Formulation, and Example," in *IEEE/ACM Transactions on Networking*, Vol. 1, No. 6 (December, 1993).

[5] Dawkins, Richard, *The Extended Phenotype.* New York: Oxford University Press, 1982.

[6] Dawkins, Richard, *The Selfish Gene.* New York: Oxford University Press, 1976.

[7] Drexler, K. Eric, and Mark S. Miller, "Incentive Engineering for Computational Resource Management," in *The Ecology of Computation*, B. A. Huberman, ed. Amsterdam: Elsevier Science Publishers, 1988.

[8] Ferguson, D.F. "The Application of Microeconomics to the Design of Resource Allocation and Control Algorithms" (doctoral dissertation).

[9] Friedman, David, *The Machinery of Freedom: Guide to a Radical Capitalism.* New York: Harper and Row, 1973.

[10] Fudenberg, Drew, and Tirole, Jean. *Game Theory.* Cambridge, MA: MIT Press, 1993.

[11] Haase, Kenneth W., Jr., "Discovery Systems," in *ECAI '86: The 7th European Conference on Artificial Intelligence* (July 1986), Vol. 1.

[12] Hamming, R. W., "One Man's View of Computer Science," in Ashenhurst, Robert L., and Graham, Susan (eds.), *ACM Turing Award Lectures: The First Twenty Years 1966-1985.* Reading, MA: Addison-Wesley, 1987.

[13] Hardin, Garrett, "The Tragedy of the Commons," in *Science* (13 December 1968) Vol. 162.

[14] Hardy, Norm, and Tribble, E. Dean, "The Digital Silk Road." To appear in *Agoric Systems: Market-Based Computation*, edited by W. Tulloh, M. S. Miller, and D. LaVoie.

[15] Harris, Jed, Yu, Chee, Harris, Britton, *Market Based Scheduling* (1987) in preparation.

[16] Hayek, Friedrich A., "Cosmos and Taxis," in *Law, Legislation, and Liberty, Vol. 1: Rules and Order.* Chicago: University of Chicago Press, 1973.

[17] Hayek, Friedrich A., "Economics and Knowledge," from *Economica, New Series* (1937), Vol. IV.; reprinted in Hayek, Friedrich A. (ed.), *Individualism and Economic Order.* Chicago: University of Chicago Press, 1948.

[18] Hayek, Friedrich A., *Denationalisation of Money*, 2nd Ed. London: The Institute of Economic Affairs, 1978.

[19] Hayek, Friedrich A., *New Studies in Philosophy, Politics, Economics, and the History of Ideas.* Chicago: University of Chicago Press, 1978.

[20] Hayek, Friedrich A., *The Constitution of Liberty.* Chicago: University of Chicago Press, 1978.

[21] Hayek, Friedrich A., *The Counter-Revolution of Science: Studies on the Abuse of Reason.* Indianapolis: Liberty Press, 1979.

[22] Hayek, Friedrich A., *Unemployment and Monetary Policy: Government as Generator of the "Business Cycle."* San Francisco, CA: Cato Institute, 1979.

[23] Hofstadter, Douglas R., "The Prisoner's Dilemma Computer Tournaments and the Evolution of Cooperation," in *Metamagical Themas: Questing for the Essence of Mind and Pattern.* New York: Basic Books, 1985.

[24] Hogg, Tad, and Huberman, Bernardo, "Controlling Chaos in Distributed Systems," Xerox PARC Technical Report #SSL–9052 (Nov. 1990).

[25] Holland, John H., Holyoak, Keith J., Nisbett, Richard E., and Thagard, Paul R. *Induction: Processes of Inference, Learning, and Discovery.* Cambridge, MA: MIT Press, 1986.

[26] Kahn, Kenneth M., and Mark S. Miller, "Language Design and Open Systems," in *The Ecology of Computation*, B. A. Huberman, ed. Amsterdam: Elsevier Science Publishers, 1988.

[27] Lenat, Douglas B., "The Role of Heuristics in Learning by Discovery: Three Case Studies," in Michalski, Rysznard S., Carbonell, Jaime G., and Mitchell, Tom M. (eds.), *Machine Learning: An Artificial Intelligence Approach.* Palo Alto, CA: Tioga Publishing Company, 1983.

[28] Lenat, Douglas B., and Brown, John Seely, "Why AM and Eurisko Appear to Work," in *The Ecology of Computation*, B. A. Huberman, ed. Amsterdam: Elsevier Science Publishers, 1988.

[29] Miller, Mark S., and K. Eric Drexler, "Comparative Ecology: A Computational Perspective," in *The Ecology of Computation*, B. A. Huberman, ed. Amsterdam: Elsevier Science Publishers, 1988.

[30] Miller, Mark S., and K. Eric Drexler, "Markets and Computation: Agoric Open Systems," in *The Ecology of Computation*, B. A. Huberman, ed. Amsterdam: Elsevier Science Publishers, 1988.

[31] Miller, Mark S., Bobrow, Daniel G., Tribble, Eric Dean, and Levy, Jacob, "Logical Secrets," in Shapiro, Ehud (ed.), *Concurrent Prolog: Collected Papers.* Cambridge, MA: MIT Press, 1987.

[32] Nagle, John B., "On Packet Switches with Infinite Storage," in *IEEE Transactions on Communications*, v. 35, no. 4, April, 1987.

[33] Nieh, Jason, Northcutt, J. Duane, Lam, Monica S., and Hanko, James G., "A Scheduling Facility in Support of Multimedia Aplications."

[34] Nisbett, Richard, and Ross, Lee, *Human Inference: Strategies and Shortcomings of Social Judgment.* Englewood Cliffs, NJ: Prentice-Hall, 1980.

[35] Rivest, R., Shamir, A., and Adelman, L., "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," in *Communications of the ACM* (Feb. 1978) Vol. 21, No. 2.

[36] Smith, Maynard J., and Price, G. R., "The Logic of Animal Conflicts," in *Nature* (1973) 246.

[37] Star, Spencer, "TRADER: A Knowledge-Based System for Trading in Markets," in *Economics and Artificial Intelligence First International Conference* (Aix-En-Provence, France, September 1986).

[38] Sutherland, I.E., "A Futures Market in Computer Time," in *Communications of the ACM* (June 1968) Vol. 11, No. 6.

[39] Tribble, Eric Dean, Miller, Mark S., Kahn, Kenneth M., Bobrow, Daniel, Abbott, C., and Shapiro, Ehud, "Channels: A Generalization of Streams," *Logic Programming: Proceedings of the Fourth International Conference*, MIT Press.

[40] Waldspurger, C. A., Hogg, T., Huberman, B. A., Kephart, J.O., and Stornetta, W.S. "Spawn: A Distributed Computational Economy." *IEEE Transactions on Software Engineering*, Vol. 18, No. 2, February 1992.

[41] Wickler, Wolfgang, *Mimicry in Plants and Animals.* New York: World University Library/McGraw-Hill, 1968.

[42] Williamson, Oliver, *Markets and Hierarchies: Analysis and Anti-Trust Implications.* New York: Free Press, 1975.

[43] Wilson, Edward O., *Sociobiology.* Cambridge, MA: Belknap Press/ Harvard University Press, 1975.

[44] Wallace, C.S. and Pose, R.D. "Charging in a Secure Environment" *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence*, Bremen, FRG, 1990. (A revised version has been published in *Security and Persistence*, Bremen 1990. J. Rosenberg and J.L. Keedy (Editors) Springer-Verlag Workshops in Computing Series. ISBN 3-540-19646-3, pp. 85-96.)