TITLE

SYSTEM AND METHOD FOR DISPLAYING MULTIPLE SOURCES OF DATA IN NEAR REAL-TIME

BACKGROUND OF THE INVENTION

Field of the Invention

This invention is in the field of computer system displays. In particular, the present invention relates to a system and method for displaying, in near real-time, data published from various computer systems and networks, such as the Internet.

Description of Related Art

The availability of powerful computer systems and networks has yielded the ability to transmit and receive a large volume of data, such as news and financial market data, from many various sources. Often, this data is critical to business operations and decision-making. For example, a stock trader may wish to obtain, almost instantaneously, news and financial data regarding a certain company or

group of companies, input that data to a financial software application, and make a trading decision based on its output. This requires a computer application that can obtain and display voluminous data from various data sources quickly and efficiently. Preferably, this application should also allow its user to filter out undesired information or expand desired information, as well as support the execution of other applications therein.

Commercial applications exist that display news, financial data and other information on computers that the applications' backers say has been "pushed" (see below for a discussion of this term) from various Internet sources, such as CNN, CNNfn, Time, Reuters, PR Newswire, Sportsticker and Accuweather, as well as local newspapers. For example, one application displays news headlines, stock information, industry updates, global weather, sports scores, etc., dynamically across the computer's screen. To obtain the information, the user executes the application and requests a download of the latest information. That information is then retrieved from the various Internet sources. The computer screen displays window buttons, each representing a different source, for example, CNN or Weather. The user selects a channel by clicking on the desired button, causing another window to display several lines of information, such as news headlines, related to the selected channel. The user may then select a single headline, and the full news story will be shown in a third window.

Another application is also said to use "push" technology, in which data is broadcast to home computers via the Internet, cable TV systems or even over-the-air signals. In this application, information is grabbed by the

computer at predictable or non-peak times, then stored on the computer's hard disk for later browsing, thus getting around slow modem connections. Although this technology does not actually speed up the modem connection, the overall browsing experience appears to be faster, because the information can be fetched more quickly from the computer's hard disk.

A third application also allows a computer client to receive information from various sources, such as Dow Jones, Reuters, Bridge, financial exchanges, the Internet, or an intranet, through either a market data platform connector, a direct data feed connector or a web connector. Prior to reaching the client computer, the data may pass through and be processed by application servers that provide financial analytics, messaging, financial modeling and graphing, which add value to the data. The client computer displays the financial data in a page of multiple windows, where each window may be resized and morphed and each page may be user-composed.

(It should be noted that none of the above-described applications is admitted to be prior art with respect to the present invention simply by being mentioned in this Background section.)

As is evident from the foregoing, the loosely-defined notion of "pushed" content as a panacea to information overload has received remarkable attention. While the rhetoric behind this technology has been impressive, current product offerings have done little to solve any real business problems. The marketplace has so far failed to deliver useful and usable applications that address specific user needs. Moreover, a closer look at popular so-called "push" products reveals fundamental weaknesses

in areas such as timeliness of content delivery, information presentation, network utilization, and application management, as follows.

First, information delivery to the above applications is not nearly as efficient or effective as it could be, even in the case where data is supposedly "pushed" from the source to the computer. This is because those applications that use so-called "push" technology are using nothing more than a timed pull. In other words, the data server is contacted periodically (typically no more than once an hour) or upon client request, and once connected, the client is responsible for downloading the updated newsfeed. In many cases, particularly when changes occur frequently and to a wide range of data, this a hugely inefficient data distribution method, wasting network bandwidth, because (1) much information is automatically pulled to the computer that is not desired by the user and never looked at, and (2) when information is updated, entire pages of information are required to be re-pulled, instead of just the incremental changes. In regard to timeliness, a timed pull system is also ineffective, because of the inability to receive near-real-time information.

In addition, even once the data has been received by the client, the display thereof is also inefficient and ineffective. Commercial displays are inefficient because of the presence of unnecessary icons and toolbars, which waste critical screen "real estate", and because the multiple windows, if available, overlap or underlap each other. Those displays are also less effective than could be desired, because they do not provide the functionality required by the user to process the data simply and quickly. For example, none of the current displays allow

a user to create easily and quickly a subwindow of desired dimension, and then run an application in that subwindow using data dragged and dropped into the window from a selected channel.

For the foregoing reasons, there is a need for a display system that avoids these problems, by making efficient use of screen space, and that permits a user to subdivide screen space into subwindows as the user sees fit, and allows the user to run applications in those subwindows, even where the applications selected by the user are not all from a single suite of applications.

SUMMARY OF THE INVENTION

Accordingly, one objective of the present invention is to address some of the above-described problems, by providing an efficient and effective display to permit a user to see data from a variety of sources quickly and easily, and use that data, if desired, in the user's or a third party's application.

Another objective is to provide a display which permits the user to create efficient subwindows, or "panes", quickly and easily, in which the user may drag and drop data from a selected data channel or run a desired application therein.

A display system receives data from one or more data sources, via a server, and displays the data from one or more of the sources, selected by a user. The user can have multiple sources active at once, and can preferably display data from at least two of the active sources simultaneously. At least one portion of the display can be subdivided by the user at will, and at being displayed

can be dragged and dropped into any of the resulting "panes". In addition, various applications can if desired be run simultaneously in respective panes, regardless of whether those applications are part of a predefined suite of applications, or not.

BRIEF DESCRIPTION OF THE DRAWINGS

These and other features and advantages of the present invention can best be understood by reference to the detailed description of the preferred embodiments set forth below, taken in conjunction with the drawings, in which:

Figures 1-4 depict four arrangements in which the present invention can be utilized.

Figure 5 is a view of an example of a display provided by the present invention.

Figures 6A and 6B are diagrams illustrating certain aspects of the process of using the display system of the present invention.

Figure 7 is a diagram illustrating certain aspects of the handling of data obtained by the display system of the present invention.

Figure 8 illustrates the model/view paradigm used in the preferred version of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

The present invention, together with other inventions developed by the assignee hereof, represents a shift

towards just-in-time information delivery. As an efficient and standards-based applications suite, those inventions provide (among other things) a ubiquitous, platform-independent interface to real-time information, such as financial data and news. This accomplishes efficient, just-in-time information delivery from the information source to the client computer via a publish/subscribe architecture and a true push server, thus offering unparalleled access to intranet, extranet, and Internet information.

In the publish/subscribe architecture developed by the assignee of the present invention, the various information sources, most likely under a contractual agreement with the push server, constantly publish and broadcast data to the push server, using any desired communication method. The client computer subscribes to desired types of information available from the push server. The subscribed information, and only the subscribed information, is sent from the server to the client. In summary, published information is pushed from the source to the server, and a subset thereof is passed on from the server to the client. Since data is constantly being pushed to the server and the client, timed pulls are not required of them. Thus, the data is always fresh, and the user does not have to wait a long time for an information download. Moreover, because only data that is desired and subscribed to by the user is communicated from the server to the client computer, the bandwidth required for the communication link between the client and server is significantly reduced. Of course, a certain amount of data in the form of subscription requests must be communicated from the client to the server, slightly increasing bandwidth requirements, but this increase is

much smaller than the reduction in bandwidth realized by not communicating unwanted information.

In addition, as soon as new data is published by any of the sources, it is sent to the server, which in turn transmits the new data to the subscribing clients. This provides for extremely rapid receipt of new information. Importantly, only changes in the data are transmitted (that is, if the new data published by the source is different only in some respects from the data already sent by the server to the client from that source, then the server does not transmit those portions which have not changed). As a result, redundant network traffic is significantly reduced, which in turn greatly reduces bandwidth between the source and the server, and between the server and the client. In summary, because the push server is a true event-driven system rather than a timed-pull server, information is delivered almost instantaneously, making network usage extremely efficient.

Using a push server also permits the automatic update of the client application software. For example, when existing netchannels (described below) need to be updated, or new netchannels become available and need to be added, the push server can push the new binary code required to accomplish these tasks directly to the client computer. The push server also supports server-side storage and retrieval of user preferences. This means that a user's profile can be referred to the server, and the data for that user customized accordingly, regardless of what computer the user employs to connect to the network. Users can log in while travelling or at home and utilize an identical desktop of applications and netchannels. The push server also can be used to manage server-based alerts. A user-specified alert (such as a stock price

warning) may also be stored on the push server, which in turn will cause the dispatching of a page or an e-mail message if the conditions of the user-specified alert are met. The push server may also specify the schema of its various data sources in real-time via XML. This enables the use of a generic client application software component to receive arbitrary data from the push server.

Several versions of such a publish/subscribe architecture are described below. In a first version, an industry-specific extranet, as shown in Figure 1, is established to provide a common information and communication network to constituent organizations. Data sources such as market data, historical data and news headlines directly feed into a push server located centrally in the extranet. Individual clients, such as the one shown in Firm A's intranet, subscribe to events generated by the extranet push server, thus eliminating the need for an in-house push server. An open API will enable the clients to subscribe to information provided by the push server. This API will handle security and entitlements for all data feeds. The individual client shown in Firm A is part of an intranet also having access to firm enterprise information. The push server may also support an interface to internal enterprise information which allows firm-wide data to appear as another data feed to client. Thus, the client software application of the present invention may display the pushed extranet information or the enterprise information or both, as will be described in more detail below.

In a second version, the push server may reside inside an enterprise's intranet for those firms whose requirements demand such, for example, Firm B shown in Fig. 2. The firm configures the internal push server in-house, which

serves clients directly. External sources feed directly into the internal push server via direct links (such as the market data) or feed indirectly via the Internet.

In a third version, shown in Fig. 3, a private push server is made accessible to the client via the Internet. In this version, there is no internal push server. Built-in authentication ensures that the push server on the Internet is accessed by authorized users only. To traverse Internet firewalls, the well-known technique of HTTP (Hypertext Transfer Protocol) tunneling is supported.

The present invention itself is directed to a display system that is particularly well suited for use with the push server arrangements outlined above. The display system, however, is not limited to use with such a push server. Thus, as shown in Fig. 4, the present invention can also be utilized when no push server is available. The client accesses the data source on the Internet by a dial-up or direct-connection. While this arrangement does not offer the benefits of a dedicated push server, the user still obtains the advantages of the display capability of the present invention, as is described in more detail below.

Overview of the Display

For this description, a knowledge of object-oriented programming, and of Java™, is assumed. In particular, familiarity and proficiency with Sun's Java™ programming language, such as version 1.1.4, released October, 1997, is assumed.

Before a full description of the display, definitions and brief descriptions of several elements thereof will be provided.

Channel: As the term is used herein, a channel is a concentration of data from a resource such as a web server, together with user-defined display preferences, such as display colors or data filters.
Channel Selector: The channel selector is the central control center of the display, and contains several icons and buttons that can be used to perform various tasks.
Montage: The montage view is a portion of the display provided by the present invention, and is implemented preferably as a JavaBean container for JavaBean-compliant components. The montage view makes it easy for the user to manage two or more applications running concurrently, and provides an attractive and effective alternative to conventional windowing environments. This is attributable to the fact that montage applications are displayed in the montage in easily configured, neatly organized areas, called "panes", rather than in the conventional overlapping, often obscured windows.
NetChannel: A netchannel is a JavaBean that can be configured to display virtually any type of dynamic content, and is also designed to support certain messaging capabilities, such as email and paging services. In the preferred embodiment, up to two visible netchannels may be displayed at once, and the user may have additional netchannels active at a given time.

The display is generated and controlled by software stored in and executed by the client computer. The display of the present invention, as shown in Fig. 5, includes two important components: the main view 500 and the montage view 506. The main view is a small floating frame for displaying information, information selection buttons, and application management. The main view contains a "channel selector" 502, and panels 5041, 504b for viewing "netchannels", which each display continuously updated

information from specific data sources. In the preferred embodiment, two netchannels can be displayed simultaneously, but the invention is not limited to this number. The main view is preferably displayed above the montage view.

The channel selector 502 is the control portion of the display, and is preferably disposed in the center of the main view. It provides graphical controls, such as virtual buttons 510 and 512 for selecting which netchannels are displayed, and for configuring the system.

Netchannels are capable of presenting virtually any type of dynamic content, including stock quotes, news headlines and special alerts. For example, one netchannel might display breaking business headlines from a major news source, while another could be used to display firm announcements relating to a specific business unit. A user might even subscribe to a netchannel configured to monitor alerts received from an enterprise computer management system.

Users may have multiple active netchannels, up to two of which may be visible at any one time in the preferred embodiment. All of this is possible because in the present invention, the source of the information is unimportant; all data is packaged into a consistent interface, as described below. Each netchannel has its own individual settings that allow the user to customize the delivered content based on personal preferences.

In the preferred embodiment, netchannels go beyond the passive display of consumer information. A complete set of alerting capabilities is also provided. These capabilities provide users with the ability to specify

various forms of notification based on any number of business events. For example, pop-up dialogs can be displayed if a competitor is mentioned in a newsfeed. A message can also be sent to a user's pager if, say, a stock falls below a certain threshold. Further, a user might click on an interesting news headline in a netchannel and view the full text of the article in a browser. This capability provides an even higher level of targeted data delivery -- the most critical events are dispatched with utmost priority.

Although netchannels are suitable for line-oriented information, some information and applications require screen-oriented information. For this, the present invention provides the montage view 506. Rather than floating each stand-alone application in its own window, the montage view contains a two-dimensional segmentable workspace in which applications can be placed. Users can subdivide this portion of the screen into rectangular segments 5081 - 508e, or "panes," of sizes appropriate to their particular needs. Panes can be resized by dragging their edges, and applications running in neighboring panes are resized accordingly. Each pane of the montage can contain a different application; applications can be added to and removed from the montage dynamically. Furthermore, if the user wishes, a particular arrangement of panes, and the particular selection of applications running in them, can be stored. During any subsequent use of the display system of the invention, the user can scroll through the stored montages. In this way, identical application layouts can be preserved from one session to the next.

The montage view 506 makes it easy to manage numerous applications running concurrently. Since there are no overlapping windows or obscured screens, this view offers

an attractive alternative to traditional windowing environments. Moreover, as shown, valuable screen space within the montage view is not wasted with the traditional title bars and sizing controls of application windows, or because of underlapping windows.

In use, data retrieved from the extranet, intranet or internet is first buffered, and the desired information is then extracted from what has been received. The client then calls the display application, which generates the main view of the display and the montage view. The creation of the main view includes creating one or more netchannels and the virtual control buttons. Creation of the montage view includes the creation of any panes the user may desire. (As mentioned, those panes may be ones designed by the user now, or may be ones the user has designed at some time in the past and stored, and has instructed be used again at this time.)

More specifically, and as indicated in Fig. 6A, when the user launches the display application, he or she is presented with a menu of existing netchannels as governed by entitlement, corresponding to contractual agreement with the channel data provider. The user may activate them or select only some for activation, and may deselect others, by means of the virtual controls provided in the main view. Up to two of the selected netchannels (in the preferred embodiment) may then be selected for display.

As shown in Fig. 6B, the user also may launch the montage view, using a virtual control in the channel selector for this purpose. If this has been done, the user may then optionally scroll through stored montage views that have been used previously. If the user finds one that he wishes to display, he activates it, and it is displayed._

If none is found, or if the user does not scroll through the stored montage views, then the user creates a new montage, by splitting the montage view (if desired), subidviding it as convenient, and using any portion of the resulting arrangement for, e.g., the enlargement of data being displayed on one of the netchannels, or to run an application.

Implementation of the Preferred Embodiment

One important feature of the invention is that the preparation of the data to be viewed is, conceptually and in fact, handled separately from the creation of the actual display views that the user sees in using the present embodiment. For this purpose, the netchannels are architected in software according to a Model/View paradigm. By segregating the data itself (the model) from its visual representations (the view), information can be presented in various forms depending on the user's preferences or needs. As will be appreciated from the following description, a single model may feed several views simultaneously. Each unique data source requires its own model. For example, models are available which gather information from particular web sites. As mentioned above, the present invention preferably also provides a generic model, that interfaces with the push server (if one is used). The client software application of the present invention stores all of these models in a "model repository". Once a model for a given data source is placed in the repository, it may be used by any netchannel which requires it.

The preferred embodiment utilizes a predefined object framework including Model, View, and Channel classes. According to the principles of object orientation, concrete implementations of channels (i.e., individual _

objects of the class channel) "inherit" from these predefined objects and implement channel-specific behavior where appropriate.

In more detail, a channel consists of several discrete parts. Each of these parts, while separate in function, contributes to a complete channel. Emphasis has been placed on creating a "clean" system architecture which separates the actual data representation of information from the visual display of information. This "Model-View" architecture allows the same information to be viewed in numerous ways, and enables a single source of data to feed multiple different views.

The first component of a channel is the Model class. Each channel utilizes at least one model. The model contains code for physically retrieving data of a particular kind from a particular source. Preferably, provision is made for both "push" and "pull" data retrieval. A particular model may, for example, be one that parses information from a given web site on the Internet, or a site on a corporate intranet, at a specified interval, or may be one that responds to dynamic events generated by a push engine. In addition to storing code for processing information, models also store user preferences information. A model that receives stock quotes, for example, will store the portfolio of stocks the user is interested in tracking. Model classes are implemented as "Observable" classes. "Observable" objects, as is well known in the art, are a type of object which generates events that may be observed, or processed, by arbitrary "Observer" objects. (This pattern allows models to generate events when new information is delivered without "knowing" anything about the objects that are interested in receiving these events.)

The second channel component is the View class. This class specifies the "look and feel" of a particular channel for display purposes. It contains instructions defining which specific graphical components, such as stock tickers or scrolling headline displays, should be used to present the channel's data. Implementations of several graphical components, or "widgets," which are programmed to respond to events from model classes, are also included. These widgets are the "observers" in the observer-observable pattern. This means that a widget can register as an observer of a model class, thus receiving events when data is changed and automatically updating its display.

The final channel component is the Channel class, which is the "glue" that binds the model and view components together. The channel class defines a default name for the netchannel and provides some other basic information such as which channel models the netchannel uses. Additionally, the channel class is responsible for actually registering each widget in that netchannel's view as an observer of those models.

Once all the pieces of a channel are bundled together, a user can subscribe to the channel and view its information. For "pull" channels, the container coordinates the refreshing of a channel by scheduling routine updates. Where a capability of handling pushed information is provided, incoming push events are also directed to appropriate "push" channels.

As a summary illustration of the foregoing, Fig. 7 shows the processing of data received from the server for a netchannel that the user has activated and designated for display. As the data is received from the server, the

corresponding model is updated to reflect the new data. The view component of each active channel that uses that model is notified that a change has occurred, and makes a corresponding change in the display.

Configuring Channels

To configure channels, the user selects an option to open up a non-resizable window where a list of available channels and selected channels is displayed. The user may select channels from an "available" list, and deselect them from a "selected" list. If desired, the system can easily be implemented such that a selection triggers a billing record to be initiated for the purpose of metering application or channel usage.

Instead of channels being added and removed as desired by the user, the "available" list may be predefined for the user based on administrator approval.

Users may order and name their selected channels as desired. Channels may be named to distinguish one instance of a view from another. Each instance may have its own configuration options.

Figure 8 depicts a typical scenario. Any model located in the model repository may be "instantiated" and provided with user preferences for a particular type of information. An example of such preferences information is a list of stocks the user is interested in viewing. The model/view architecture allows for an arbitrary number of views to register as "observers" of the model. Because the model is concerned only with retrieving information from the corresponding data source, it is unimportant what kind of object any given observer is. As new or updated information becomes available (e.g., a change in stock

price) each registered observer is automatically notified of the change.

The client software application preferably contains a core suite of netchannels useful for presenting information from prepackaged data sources. However, because an organization may also require the ability to create its own netchannels, several netchannel development options are preferably also supported by the client software application. For basic netchannels, a "View Templates" software application is preferably provided including several templates that define netchannel layouts that are commonly found to be convenient. Users can use these templates to build new netchannel views with minimal coding effort. For example, a particular template might specify the location of a corporate logo, stock ticker, and news headline ticker. The user connects each element in the netchannel view to an existing model from the model repository. Once connected, the new channel is then ready to use. In the event that the model repository does not contain a model suitable for the new netchannel, the client software application provides a custom Java™ Application Programming Interface (API) that facilitates information retrieval. Models developed according to this API can be added to the model repository and used in subsequently-created netchannels.

Another option is preferably provided for creating a custom netchannel. A GUI "ChannelBuilder" software application offers and advanced drag-and-drop interface for developing highly customized netchannels with no coding effort. This application allows developers to select from a palette of available components and add these elements to channels in real-time. The GUI ChannelBuilder also displays a hierarchical view of all

available data sources in the model repository. Channel designers simply draw a line from a data source to a visual component, and the component is connected to that source. Channels can be tested in this environment to ensure proper functionality, and the GUI ChannelBuilder has the capability of generating a compiled channel package with the press of a button.

Data Acquisition

The invention provides a full set of data services to channel models. A "Model Services" component abstracts many of the common functions required by channels. For example, Model Services contains code that retrieves data from web pages. By using Model Services as opposed to raw Java™ calls to perform this task, developers of channels need not write cumbersome network access code. Moreover, Model Services automatically detects if a proxy server is installed and modifies the network request accordingly. Additional functions provided by Model Services are subscribing and unsubscribing to "push" topics and/or events; launching a web browser to display an HTML document; notifying the container that the channel is "on alert" and should be displayed; and sending a message to a user's beeper. Aside from the obvious benefit of reducing tedious programming work, the availability of Model Services means that users do not have to input preferences information such as a pager PIN or web browser location for each channel application; user data is shared among all applications.

Pull-type channels request Model Services to retrieve a web resource on a regular basis, ranging from every few minutes for rapidly changing data such as stock quotes, to only once every few hours for less dynamic data. Often this interval is user-configurable. Model Services will

take the simple request from the channel and formulate a proper HTTP request for a web server. If the user is operating behind a firewall, Model Services modifies the request to go via the proxy server. Data is returned to the channel in the form of HTML (Hypertext Markup Language). Channel models then parse this data using custom routines, or utility classes provided as part of the display system which aid in processing HTML documents.

Push channels, on the other hand, use Model Services to subscribe to a particular data topic. Model Services takes the request and formats a subscription request for the push engine in use. When new data for this topic is generated, it is sent to the container, which determines the channel or channels for which the data is destined and sends the appropriate message. Because pushed data is formatted according to a special protocol, no parsing of HTML is required in this scenario. The models used in these cases simply add the content of the event to their data store and notify all observers that new content is available.

The process by which a piece of information is retrieved and ultimately displayed to the user is different depending on whether the netchannel in question is a "pull" or "push" channel. Both processes are described below.

Pull channel models are implemented as Java™ threads and are timed to retrieve the complete contents of a raw resource (information source) at a given interval. This resource may be any number of file types, such as an HTML document on the World Wide Web, or a preformatted text file located on a corporate intranet.

Pull models are programmed to retrieve resources from a well-defined location, and contain specific rules for parsing information that is retrieved from this particular site. In most cases this parsing involves scanning an HTML document and extracting only the relevant information to be displayed. Oftentimes information deemed relevant in an HTML file is in some preformatted state (e.g., bullet points in a web document), so the model will simply scan for each instance of the HTML code that formats the relevant information. For example, a model which retrieves news headlines might look for each instance of the HTML <LI> tag, which signifies a bullet point, and extract the text immediately following the tag for display as a news headline. Model Services is utilized to automatically retrieves the contents of a given resource via a simple method call.

Unlike inherently static pull models whose contents are reset during each update, push models are dynamic and maintain a constantly changing state throughout the life of the application. When a push model is created, it registers interest in a particular topic with the push server in use. This registration process is accomplished via Model Services and notifies the push engine that a netchannel model wishes to receive events related to a particular topic, such as a change in a stock price or breaking news on a specific company. Since all communication with the push server is conducted via Model Services, a unique ID code is assigned to the channel model at the time of subscription for purposes of tracking incoming and outgoing messages.

When information changes for a topic to which a push model has subscribed, the push server sends a message to Model

Services containing the new data. This message contains only the data that has changed; that is, the minimum information necessary to update the information the net channel already has, is transmitted. Model Services references the ID code attached to the incoming message and forwards the data to the corresponding model. A predefined data structure is utilized for the update information being sent, and therefore, no parsing of information is required. All relevant data is preformatted for the netchannel models.

Regardless of how data is obtained (via push or pull methods), the subsequent processing is identical. A library of "netchannel widgets", which are special objects programmed to listen for events generated by netchannel models, is provided. These widgets are graphical information controls, such as stock tickers, scrolling headline displays, and dynamic text labels. In much the same manner that a push channel mode subscribes to events form a push server, netchannel widgets register interest in updates from netchannel models at the time the user makes a particular netchannel active.

One example of what a model can do is as follows. As a netchannel model processes new information, a data structure is created which contains all relevant information for netchannel widgets. This structure contains the text of the data to be displayed, an optional URL which links to more information related to the data, a foreground and background color for the information display, and a timestamp. Each model automatically broadcasts these structures to all widgets that have previously registered interest in that model's updates. As the updates are received, the widget adds the new data to its information display and shows the data onscreen.

Filtering

It is preferable to provide granular filtering of information channel-by-channel, although this is not itself part of the display system of the present invention. Each channel contains options for specifying the type of information a user is interested in within the context of the channel's content. An example would be filtering a financial news channel to display only news relating to stocks in the user's personal portfolio. Individual customization screens allow the user to create filters for each channel. Multiple filters may be created for each channel, and can be saved and re-used at will.

In addition to its filtering capabilities, the system is capable of generating user-defined alerts. In much the same manner that filters are defined, users may establish criteria for special alerts. A typical scenario would involve an alert when a particular stock's price hits a specified level, or when a topic of interest to the user is reported in the news. An appropriate interface is provided for building complex alert criteria and storing them along with other preferences.

When the system is used in a "push" environment, both filters and alerts are stored on the remote push server, enabling user-specific settings to follow the user to whatever computer she is presently using. In the absence of a push server, filters and alerts can be stored on a local machine.

Serialization

The invention utilizes the concept of "serialization" to preserve application state and provide continuity from one session to the next. All user-specific information, including preferences, filters, alert criteria, and settings, are "serialized" at the end of each application execution. The serialization process involves taking a "snapshot" of current memory and saving it to an arbitrary output stream. This output stream may be a local disk file, or a socket connection to a push server.

By storing this complete snapshot of memory, all user preferences and configuration information is easily stored and restored. Individual channel developers need not be concerned with storage of user settings since the container automatically handles this operation. Moreover, since complete state information can be serialized to a push server over a network connection, the same configuration a user enjoys on his desktop computer by day can be accessed from home on his laptop.

As described above, the invention also allows for a montage view presentation of applications. Because each model is preferably implemented using Sun's Java™ programming language as a JavaBean, its properties can be discovered dynamically. This eliminates the need to develop custom settings screens for each model. Applications are discovered and made available to the user to place into the segmentable montage workspace. Since each application is a JavaBean component, the viewing and modification of attributes, or "properties," is supported. In addition, the top-level montage view container itself is a JavaBean, which means that the exact subdivision of the screen into panes, as well as the applications which reside within these panes, are all part of a large

containment hierarchy. When the user exits the application, the top level of this hierarchy is serialized, and as a result, each application contained within the hierarchy is serialized as well. Later when the application is restarted, the desktop appears exactly as it was left off. Like channels, montage view applications need not provide code for storing and retrieving user preferences, as the container automatically handles this.

It is preferable that the channel selector, the netchannels and the montage view are implemented in software as JavaBeans. The Sun JavaBeans specification defines a software component model for Java™ that supports the creation of reusable components supporting a common interface. Important benefits of JavaBeans include introspection, serialization and reuse. In regard to introspection, users of JavaBeans do not need to know the properties and methods supported by the Bean at design time. Built-in introspection methods are used to discover dynamically the interface of a particular component, thus eliminating the need to restrict functionality of components to a lowest common denominator, or to provide specialized handling in code. By discovering a particular netchannel's properties, for example, the present invention is able to build a configuration screen for that netchannel on the fly. The JavaBeans specification also supports automatic serialization of components, which means that state information such as user preferences can be stored and retrieved without writing specialized code for each Bean. This provides an efficient method for maintaining settings strong among various complex components. In regard to reuse, because JavaBeans are components with well-defined interfaces, JavaBeans of various types can be instantiated easily regardless of the

container. This means that a particular application can be contained in the montage view, a floating window, or in any other compatible container. If Java™ is used, both the main view and the montage view are JavaBean containers. In the main view, both the left and right views hold netchannel beans, whereas each pane in the montage view may contain a JavaBean application. An additional benefit of implementing the montage view as a JavaBean container is that third-party JavaBean components may be instantiated in a montage pane, thus providing the significant benefits of the montage view to additional applications.

Preferably, the client software applications are implemented as Java™ applications rather than Java™ applets. (Applications are standalone programs that are executed directly by the operating system, while applets are Java™ programs that run within a web browser.) A major benefit of Java™ applications is the ability to use Sun's reference Java™ implementation. (Although ostensibly all versions of Java™ Virtual Machines (VM's) are functionally equivalent, practical experience reveals differences among various vendors' VM's that are significant enough to affect performance.) Also favoring Java™ applications is the fact that the major browser vendors have yet to update their browsers to the most recent Java™ specification. Without capabilities introduced in Sun's October 1997 Java™ release (version 1.1.4), key functionality is not possible. Although most of the functionality of the client software application can be implemented with pure Java™ code, a small number of key functions are only available via platform-specific function calls.

To address these issues and preserve all cross-platform functionality, commonly used platform-specific calls are abstracted in a common Java™ class library. Rather than having application code call a system native method, the client software application contains functions such as "loadWebDocument" which will perform the platform-specific action regardless of the user's operating system. By replacing this single class library according to the operating system in use, no application contains platform-specific code.

One example of this is hyperlinking from a netchannel headline to a web document. Java™ has the capability to launch an instance of any application, including a web browser, but cannot independently send the platform-specific request to load a web page in the browser. In this specific example of launching a web browser, another alternative is provided. A Java™-based web browser application is preferably provided as part of the display system of the invention. This browser runs in the montage view and can optionally display pages when users click on netchannel links. Although this browser does not support all advanced web features, it does provide sufficient rendering of most pages.

Upon startup, a well-known directory containing channel binary files is scanned. In the preferred implementation, all channels are implemented as JavaBeans and stored in Java™ Archive (JAR) files. The main code reads each JAR file in this special directory and discovers any channels that are contained within the JARs. Once the channels are discovered, they are automatically queried for basic information such as the title of the channel, and a small graphical icon representing the channel. This information

is cached and used later to allow the user to add and remove active channels.

Each JAR file is "stamped" with a code indicating the version of the particular channel application(s) it contains. New versions of channel JAR files may be stored on a remote server for the purpose of automatic software upgrades. At startup, this remote server is queried and determines if newer versions of any applications exist. This process is accomplished by comparing the version stamp of the local JAR file with the version reported by the server. If a newer version exists, the user is prompted to upgrade, and if the decision is made to do so, the new JAR file (or files) is automatically downloaded from the server to the well-known directory. In this manner, applications software is automatically upgraded with minimal user intervention. System administrators may optionally force upgrades without explicit user acceptance, thus providing powerful central administration of software distribution. Any errors that occur during the automatic upgrade process can be logged and analyzed. In addition to new versions of existing channels, totally new channels may be sent to client systems in the same manner.

For its graphical displays, the present invention preferably employs the known Netscape Internet Foundation Classes (IFC) user interface components. IFC is a Java™ user interface application framework for building applications independent of operating system-specific windows and user interface controls. The foundation classes enable development of rich, professional user interfaces that are not affected by individual platform idiosyncrasies. IFC also has close ties to the next-generation Java™ Foundation Classes (JFC) which are

presently under joint development by Sun, Netscape and IBM. JFC, an outgrowth of IFC, provides a "pluggable" look and feel for Java™ applications which enables users and developers to select the graphical user interface of their choice for each application. JFC will support all major IFC features, including lightweight user interface components and IFC's basic application framework. Since JFC will become a part of the core Java™ package when released, the use of IFC will allow for the smoothest possible upgrade to JFC.

As described above, the present invention preferably supports the use of several applications. All of these applications can be hosted in the montage view. Such applications are also preferably developed with Sun's Java™ programming language and as JavaBeans. Consequently, every component in the display system of the present invention, and the applications used therewith, are platform-independent and web-enabled, providing tight integration with all popular browsers, where appropriate.

One such application is a "market minder" which is a table-based display of securities in a user's portfolio. Prices, bid/ask values, volume, and other data are updated continuously for each item in the portfolio. Multiple paging, e-mail and pop-up alerts can be associated with events specific to each security. When operating with a push server, the market minder application can also display intraday and historical price/volume charts for any security. Charts can be "drifted-down" to show more details for a given time period. Another application is a "news minder", which aggregates news headlines related to user-defined topics from various sources. These topics can be simple keywords, complex queries, or the companies represented in the user's market minder portfolio.

Selecting a topic displays a list of all current headlines with summaries, original source, and date of posting. Users can click on a particular headline and view the full article in rich text, complete with graphics and hyperlinks. Additionally, intelligent filters can be applied on a topic-by-topic basis, enabling further refinement of information on topics that contain a large amount of news. A third application is a "unified inbox". Traditionally, users are required to use separate applications for e-mail, voicemail, and faxing. The unified inbox wraps all three types of incoming messagings into a single interface. Users can read e-mail, listen to voicemail, and view faxes all from the same unified inbox application. Other, smaller utility applications, such as financial calculators and graphing utilities, may also be invoked within the montage view.

Of course, it will be appreciated that the invention may take forms other than those specifically described. The scope of the invention, however, is to be determined solely by the following claims.

WHAT IS CLAIMED IS:

1. A method for generating a display for a computer system comprising the steps of:

receiving, from a server, data from one or more sources of data, to which sources the computer system subscribes;

generating a display image partitioned into a plurality of display areas;

selecting for display at least one source of data from among the one or more sources received in said receiving step; and

displaying the selected at least one source of data in a corresponding partitioned display area.

2. A method according to Claim 1, further comprising the step of partitioning one of the partitioned display areas into a plurality of subpartitioned display areas.

3. A method according to Claim 2, further comprising the steps of selecting the at least one source of data displayed in the corresponding partitioned display area, dragging the selected data from the corresponding partitioned display area, and dropping the dragged data into one of the subpartitioned display areas.

4. A method according to Claim 1, further comprising the step of running an application in another one of the partitioned display areas.

5. A method according to Claim 2, further comprising the step of running an application in one of the subpartitioned display areas.

* C L M S *

6. A method according to Claim 5, further comprising the step of running a second application in another one of the subpartitioned display areas.

7. A method according to Claim 6, wherein the applications are two parts of a single predefined suite of applications.

8. A method according to Claim 6, wherein the applications are not parts of a single predefined suite of applications.

9. A method according to Claim 1, further comprising the step of defining display conditions for execution of said displaying step.

10. A computer system, comprising:

means for receiving, from a server, data from one or more sources of data pushed to said computer system from the server, the sources of data being ones subscribed to by said computer system;

a display for displaying an image;

means for partitioning the image of the display into a plurality of display areas; and

means for selecting for display at least one source of data from among the one or more sources received by said receiving means, said display displaying the selected at least one source of data in a corresponding partitioned display area.

11. A computer system according to Claim 10, further comprising means for partitioning one of the partitioned display areas into a plurality of subpartitioned display areas.

* C L M S *

12. A computer system according to Claim 11, further comprising means for selecting the at least one source of data displayed in the corresponding partitioned display area, dragging the selected data from the corresponding partitioned display area, and dropping the dragged data into one of the subpartitioned display areas.

13. A computer system according to Claim 11, wherein said selecting means comprises one or more virtual buttons, each of said virtual buttons corresponding to one of the sources of data.

14. A tangible computer-readable medium storing instructions which, when loaded into a programmable apparatus, cause that apparatus to perform a method comprising the steps of:

receiving, from a server, data from one or more sources of data, to which sources the computer system subscribes;

generating a display image partitioned into a plurality of display areas;

selecting for display at least one source of data from among the one or more sources received in said receiving step; and

displaying the selected at least one source of data in a corresponding partitioned display area.