03-23-01

Approv

J1031 U.S. PTO 03/21/01

Please type a plus sign (+) inside this box➔ [+]

modified PTO/SB/16(8-00)

PROVISIONAL APPLICATION FOR PATENT COVER SHEET

This is a request for filing a PROVISIONAL APPLICATION under 37 CFR 1.53(c).

jc979 U.S. PTO 60/278303 03/21/01

Docket Number:	22726-05901

INVENTOR(s)

Given Name (first and middle [if any])	Family Name or Surname	Residence (City And Either State Or Foreign Country)
Timothy	Tuttle	San Francisco, California
Robert S.	Dreyer	San Francisco, California
Karl E.	Rumelhart	Palo Alto, California

☐ Additional inventors are being named on ___ separately numbered sheets attached hereto.

TITLE OF THE INVENTION (500 characters max.)

NODE SPECIALIZATION ARCHITECTURE

CORRESPONDENCE ADDRESS

Direct all correspondence to:

☒ Customer Number 00758 →

ENCLOSED APPLICATION PARTS (check all that apply)

☒ Specification	*No. of Pages:*	85	☒ Return Postcard	
☒ Drawing(s)	*No.of Sheets:*	3	☐ CD(s), Number	
☒ Application Data Sheet See 37 CFR 1.76			☐ Other *(specify)*	

METHOD OF PAYMENT (check all that apply)

☒ Applicant claims small entity status. See 37 CFR 1.27

☒ Fee Transmittal Form Enclosed (in duplicate) ☒ Check Enclosed

The invention was made by an agency of the United States Government or under a contract with an agency of the United States Government.

☒ No.

☐ Yes, the name of the U.S. Government Agency and the Government contract number are: _____

SIGNATURE OF ATTORNEY OR AGENT

Signature:			
Attorney/Reg. No.:	Brian M. Hoffman, Reg. No. 39,713	Dated:	March 21, 2001

CERTIFICATE OF MAILING

I hereby certify that this correspondence is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service pursuant to 37 CFR 1 10 in an envelope addressed to: Box Provisional Patent Application, Commissioner for Patents, Washington, DC 20231 on the date shown below.

Signature:			
Typed or Printed Name:	Brian M. Hoffman	Dated:	March 21, 2001
Express Mail Mailing Number:	EL550795571US		

USE ONLY FOR FILING A PROVISIONAL APPLICATION FOR PATENT

H9000/01162/SF/5044873.1

EXPRESS MAIL NO. EL550795571US

PTO/SB/17 (10-00)(modified)
Approved for use through 09/29/01, OMB 0651-0032
Patent and Trademark Office: U.S. DEPARTMENT OF COMMERCE

0002/PTO(modified)
Rev. 10/2000

U.S. Department of Commerce
Patent and Trademark Office

FEE TRANSMITTAL

TOTAL AMOUNT OF PAYMENT

Subtotal (1) + Subtotal (2) + Subtotal (3) = **($) 75.00**

Complete if Known	
Application Number	NEW
Filing Date	HEREWITH
First Named Inventor	Timothy Tuttle *et al.*
Group Art Unit	UNASSIGNED
Examiner Name	UNASSIGNED
Attorney Docket Number	22726-05901

METHOD OF PAYMENT

1. The Commissioner is hereby authorized to:

☐ Charge the indicated fees to the below mentioned deposit account.

☒ Charge any additional fee required under 37 CFR 1.16 - 1 21 or credit any over payments to the below mentioned deposit account. †

☒ Applicant claims small entity status See 37 CFR 1.27

Deposit Account Number: 19-2555
Deposit Account Name: FENWICK & WEST LLP

A Duplicate Copy of this authorization is attached

2. ☒ **Payment Enclosed:**
☒ Check ☐ Credit Card ☐ Other

FEE CALCULATION (fees effective 10/01/2000)

1. FILING FEE

Large Entity Fee Code/Fee	Small Entity Fee Code/Fee	Fee Description	Fee Due
101/$710	201/$355	Utility Filing	
106/$320	206/$160	Design Filing	
108/$710	208/$355	Reissue	
114/$150	214/$75	Provisional Filing	75

SUBTOTAL (1) **($) 75**

2. CLAIMS

Large Entity Fee Code/Fee	Small Entity Fee Code/Fee	Fee Description
103/$18	203/$9	Claims in excess of 20
102/$80	202/$40	Independent claims in excess of 3
104/$270	204/$135	Multiple dependent claim
109/$80	209/$40	Reissue independent claims over original patent
110/$18	210/$9	Reissue claims in excess of 20 and over original patent

FEE CALCULATION (continued)

3. ADDITIONAL FEES

Large Entity Fee Code/Fee	Small Entity Fee Code/Fee	Fee Description	Fee Due
105/$130	205/$65	Surcharge - late filing fee or oath	
127/$50	227/$25	Surcharge-late provisional filing fee or cover sheet	
147/$2,520	147/$2,520	For filing a request for reexamination	
115/$110	215/$55	Extension for response within first month†	
116/$390	216/$195	Extension for response within second month†	
117/$890	217/$445	Extension for response within third month†	
118/$1,390	218/$695	Extension for response within fourth month†	
128/$1,890	228/$945	Extension for response within fifth month†	
119/$310	219/$155	Notice of Appeal	
141/$1,240	241/$620	Petition to revive unintentionally abandoned application	
142/$1,240	242/$620	Utility Issue Fee (Or Reissue)	
143/$440	243/$220	Design Issue Fee	
122/$130	122/$130	Petitions to the Commissioner	
126/$180	126/$180	Submission of Information Disclosure Statement	
179/$710	279/$355	Request for Continued Examination (RCE)	
581/$40	581/$40	Recording each patent assignment per property (times number of properties)	
146/$710	246/$355	Filing a submission after final rejection (37 CFR 1.129(a))	
149/$710	249/$355	For each additional invention to be examined (37 CFR 1 129(b))	
	Other fee *(specify)*		
	Other fee *(specify)*.		

SUBTOTAL (3) **($) 0**

For	(Col. 1) No. of Existing Claims		(Col. 2) Highest No. Previously Paid For		(Col. 3) Extra**		Fee		Fee Due
TOTAL		minus*	20 or 0	=		x		=	
INDEP		minus*	3 or 0	=		x		=	
[ ] First presentation of multiple dependent claim								=	

* *Subtract the greater number of Col. 2*
** *If the difference between Col. 1 and Col. 2 is less than zero, then enter "0" in Col. 3*

SUBTOTAL (2) **($) 0**

SUBMITTED BY		Complete (if applicable)	
Typed or Printed Name	Brian M. Hoffman	Reg. Number	39,713
Signature		Date	March 21, 2001

† *Request for Extension of Time per 37 CFR 1 136 (a)(3) made hereby*

Rev 09 30 2000
H9000/01162/SF/5044875.1

INVENTOR INFORMATION

Inventor One Given Name:: Timothy
Family Name:: Tuttle
Postal Address Line One:: 3449 22nd Street, Apt. 8
City:: San Francisco
State or Province:: California
Country:: United States
Postal or Zip Code:: 94110
City of Residence:: San Francisco
State or Province of Residence:: California
Country of Residence:: United States
Citizenship Country:: US
Inventor Two Given Name:: Robert S
Family Name:: Dreyer
Postal Address Line One:: 3918 23rd Street
City:: San Francisco
State or Province:: California
Country:: United States
Postal or Zip Code:: 94114
City of Residence:: San Francisco
State or Province of Residence:: California
Country of Residence:: United States
Citizenship Country:: US
Inventor Three Given Name:: Karl E
Family Name:: Rumelhart
Postal Address Line One:: 1244 S. California Avenue
City:: Palo Alto
State or Province:: California
Country:: United States
Postal or Zip Code:: 94306
City of Residence:: Palo Alto
State or Province of Residence:: California
Country of Residence:: United States
Citizenship Country:: US

CORRESPONDENCE INFORMATION

Correspondence Customer Number:: 00758
Fax One:: (650) 494-1417
Electronic Mail One:: bhoffman@fenwick.com
Fax Two:: (415) 281-1350

APPLICATION INFORMATION

Title Line One:: Node Specialization Architecture
Total Drawing Sheets:: 3
Formal Drawings?:: No
Application Type:: Provisional

Docket Number: 22726-05901
Secrecy Order in Parent Appl.?:: No

REPRESENTATIVE INFORMATION

Representative Customer Number:: 758
Registration Number One:: 42120
Registration Number Two:: 39713

Source:: PrintEFS Version 1.0.1

Node Specialization Architecture

Background

When designing a real-time routing network, one option is to distribute every message to every node. If a node is servicing viewers who have registered to receive that message, it will forward it to those viewers. Otherwise, the node will ignore the message.

Under certain circumstances, it may be desirable to not send every message to every node. However, if instead we send some messages to some nodes and some messages to other nodes, we encounter (at least) two problems.

1. For a given message, how do we decide which nodes should receive it? and
2. How do we handle/prevent a viewer wanting messages that its node is not receiving?

In this invention, we describe several technologies for addressing these problems.

Description of the Invention

METHOD 1: Registering at Multiple Levels

In this method, the decision about which nodes receive which messages is dynamic. Based on the viewers it is servicing, a node registers with each gateway which messages it needs. In other words, just as the node keeps track of which viewers want which messages and sends each viewer only messages for which it has registered, in this approach the gateways keep track of which messages each node wants and forward to each node only messages for which it has registered. A node "wants" a message if at least one of its viewers wants the message.

To be more precise, when a viewer registers or deregisters an ID with a node, the following occurs:

(a) The node updates its table of registrations appropriately.
(b) The node determines whether the viewer registered any IDs that were not previously registered on that node, or deregistered any IDs for which it was the last interested viewer. In either case, there was a change to what this node "wants." If so, go on to part (c)
(c) The node informs all gateways of the change in what IDs it wants.
(d) The gateways update their tables of registrations appropriately.

When a gateway receives a message, the following occurs:

(a) The gateway looks up in its table of registrations to determine which nodes should get the message. It then forwards the message to those nodes.

(b) Each node that receives the message looks up in its table of registrations to determine which viewers should get the message. The nodes then forward the message to those viewers.

Implementing this technology is fairly straightforward since it primarily involves adding node functionality to the gateway and viewer functionality to the node.

This method addresses the two problems identified above: 1. Messages are sent to nodes that have registered the relevant ID, and 2. No matter which node a viewer connects to, that node will receive all messages that the viewer wants.

There are a number of alternative embodiments of this technology. For example, while the method described involves registration at two levels, gateways and nodes, the concept extends to registration at any number of levels. Also, rather than deregistering an ID every time it disappears from the node's table, the node could wait for a period of time or until some event occurs, before deregistering it. (This alternative would reduce the number of registration changes required between nodes and gateways.) For example, it could wait until it receives a message associated with that ID. Alternatively, the wait time may be fixed, random, or statistically driven based on the frequency of registrations for that object ID, or registrations for the same information provider as the objects whose ID is being deregistered (this latter approach preserves the objects from more frequently used information providers in the registration tables longer).

METHOD 2: Viewer Proxies

In this approach, we split the space of possible messages into some number of, say N, categories. We also designate N types of nodes. When a gateway receives a message, it first determines which category the message is in. If the message is in category x, the gateway forwards it to the nodes of type x. (Alternatively, a more complicated mapping between message categories and node types is possible. For example, messages of category 1 could be send to nodes of type 1, 2, and 3 whereas messages of category 2 are sent to nodes of type 3 and 4 etc.)

In what follows, we first discuss how messages can be split into categories. Then we address a challenge raised by this situation, namely how a viewer can receive messages from multiple categories.

Message Categories

The task of splitting messages into categories is equivalent to the task of choosing a function from the space of allowable pairs, (information provider ID, object ID), into integers between 1 and N. In practice, this function will be some combination of table-lookups and hash functions.

In one embodiment, we categorize messages based entirely on a table lookup. For example, we assign information providers to different categories. When a message arrives, the gateway looks up the information provider ID in a table that indicates which node type (or types) should get the message. Alternatively, to the extent to which the object IDs are known ahead of time, we may do the same thing based on object IDs. In one approach, object IDs could be pre-pended with the type of node to get the message. For example, messages associated with the object ID "Bang13someString" would be forwarded to nodes of type 13.

In another embodiment, we split messages into categories using a pure hash function. For example, we define a hash function that maps pairs (information provider ID, object ID) → integers between 1 and *N*. When a message arrives, we compute the hash function of its IDs. If the result is *x*, we forward the message to nodes of type *x*, etc.

Of course, we may use a combination of table lookups and hash functions. For example, certain information providers' messages would be sent to a special node type based on a table lookup, while the others messages are distributed among the remaining nodes using a hash function. Another possibility is to use a table lookup to find a group of node types and use a hash function to distribute messages among them. Many other configurations are possible. One general approach is to incorporate a priori knowledge (such as that certain message types will be very popular based, e.g., on the source information provider) in table lookups and, when no a priori knowledge is available, distribute messages as evenly as possible using hash functions.

The techniques described above are straightforward to implement and do not introduce a significant burden on network resources. They are also highly flexible – by simply changing the function used to categorize messages, entirely different forwarding behaviors can be created. This provides a mechanism for "tuning" network performance.

Message categorization provides a solution to the first problem identified above, namely how to decide which nodes get which messages. Now we turn to the second problem: how to ensure that viewers have access to the messages they need.

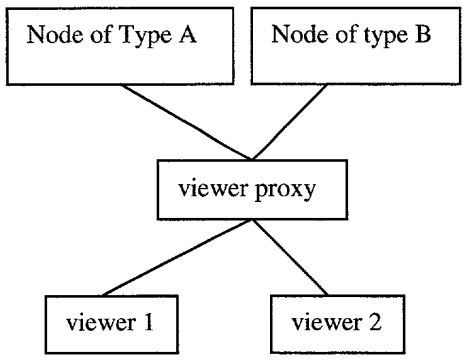
Messages to Viewers

Here we present technologies to handle/prevent the following situation: a viewer connects to a node of type *x*. However, the viewer wants to receive messages of type *y* which are not forwarded to nodes of type *x*. Since the node never receives the messages, neither will the viewer.

The first approach is to ensure that each viewer only registers IDs from one category. For example, if we categorize messages by information provider ID, we could require that each viewer only register for messages from one information provider. This can be done, for example, by configuring the load balancer to ensure that viewers registering for messages from one information provider would connect to a node of the type that receives messages from that information provider.

If we wish to give viewers complete freedom in their registrations, we should use a different approach – we should allow a viewer to communicate with nodes of several types. One way to do this is for the viewer to make multiple connections to the network. More precisely, if the message-categorization functionality were built-in to the applet (client component), before registering IDs, the applet could determine which categories of messages it needed and so which types of nodes it should connect to. It would then make a connection to one node of each relevant type, registering the corresponding IDs.

An alternative to asking a viewer to maintain multiple open connections is for the network to establish a kind of proxy service. I will describe several variations of this concept, differing primarily in where the viewer registration information is stored.



In the first variation, viewer registration information is stored at the nodes. The idea is that, instead of connecting directly with one or more nodes, a viewer connects to a viewer proxy, which is in turn connected to a node of each type. When a viewer wants to register a particular ID, the proxy determines which type of node handles that ID and forwards the registration request to that node. The proxy should also include an identifier of its own connection to the viewer, such as a pointer to the socket for that viewer connection. When the node wants to direct a message to the viewer, it sends it to the proxy along with the pointer to the viewer socket. When the proxy gets the message, it simply pushes it onto the socket.

So, whereas without node specialization, each node maintains a hash table in which a pair (information provider ID, object ID) is associated to a list of pointers to sockets for viewer connections, now the node hash table will associate to the pair (information provider ID, object ID) a list of pairs (pointer to socket for viewer proxy, pointer to socket for viewer *on the proxy*).[†]

All of this is a fancy way of saying that the registration goes roughly like this: the proxy says to the node, "This guy, whom I refer to as 'Al', wants to register something with

[†] A slight modification would be for the node to associate a pair, (information provider ID, object ID), with a list of pairs (pointer to socket for viewer proxy, LIST of pointers to sockets for viewers on the proxy). The node would pass a message to the proxy along with a list of all the sockets it needs to forward it to. This prevents a node from having to send the same message to the same proxy multiple times because there happen to be multiple viewers on that proxy who need it.

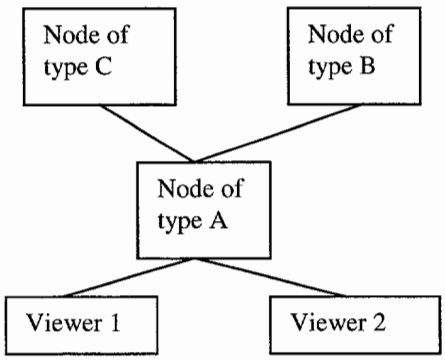
you." The node records in its table that "If I get any info about the Superbowl score, I should send it to the proxy I call 'Jim' and tell him to send it to the guy he calls 'Al'."

One advantage of this scheme is that it reduces memory loads. The proxy doesn't have to store any state, and each node has a restricted list of object IDs that it could possibly have to deal with. Another advantage is that, since the proxy has to do very little processing to forward a message, this approach is very fast, even though it introduces another layer in the system.

In the next variation, the viewer proxy is responsible for keeping track of object registrations. (In this scenario the "viewer proxy" might reasonably be considered a "node," but for simplicity we will stick to our original language.) As before, a viewer connects to a viewer proxy that is itself connected to a node of each type. The viewer proxy records the viewers' object registrations. The proxy also registers with the nodes all objects for which it needs to receive updates, registering objects of type *x* with the node of type *x* etc. From the point of view of a node, a viewer proxy is just like a "viewer." When a node receives a message, it looks up in its table which proxies have registered that ID and passes the message to them. When a proxy receives a message it looks up in its table which viewers want the message and forwards it to them.

This scheme has certain things in common with Method 1 described above. One difference is that a proxy registers a given object with only one node, rather than with each gateway. Moreover, the set of objects a single node stores state about is limited to the category of objects that it handles.

In the final variation on the proxy idea, the viewer proxy is not a separate box but rather a function adopted by the nodes themselves.



When a viewer connects to a node, the objects it needs to register that happen to be of the type handled by that node are registered with that node as usual. For the other objects, the node acts as a proxy in either of the two senses described above. That is, either the registration is passed to a node of the appropriate type along with an identifier of the

viewer connection, or the node itself "registers one level up," which in this case means with another node.

This scheme can be set up so that either there is one preferred type of node that always serves as the proxy, or each node type can serve as both a node and a proxy. For example, we could designate one node type to handle the most common IDs. In this way, the proxy functionality would be utilized relatively rarely. On the other hand, if each type of node can serve as a proxy, we could include some intelligence in the applet that determines which object category predominates for that viewer and requests a connection to a node/proxy of that type. As before, the goal would be to minimize the need for bringing the proxy functionality into play.

To summarize, to implement method 2:

(a) Split the space of possible messages into categories and designate different "types" of nodes to handle the different categories, and
(b) Establish a mechanism whereby a single viewer can communicate with nodes of more than one type, either by helping a viewer establish multiple connections to the network, or by connecting a viewer to a proxy which is itself connected to nodes of different types.

Introduction

This document presents the technical specifications for the monitoring and logging components of the Bang Network that are to be included within the beta release of late November. Detailed functional descriptions of the components can be found in the Monitoring Design Proposal document.

Components and Interfaces

MonitoringAgent

The MonitoringAgent (MA) holds a set of registered listeners. Each listener is a subclass of the abstract class AgentModule, and all AgentModules implement the MonitoringAgentListener interface. MAs themselves also implement the MonitoringAgentListener so that MA to MA communication is enabled. MAs are remote-enabled through Voyager, so that remote communication is seamless.

MonitoringAgents and their modules must be started in an independent process on each box in which they reside. These boxes include each node, each gateway, each appServer, and a central box per cluster (currently named Charon at Colo.com). This process will be a Java class that configures the MA and modules based on a configuration file.

System components (NodeServer, GatewayServer, etc) will connect with MonitoringAgents through MonitoringAgentProxys (MAPs). These proxies will implement the same interfaces as MA so they are virtually hidden. MAPs are important so that we can catch Exceptions from Voyager (see the section on fault tolerance). This communication is achieved via the loopback port so as not to involve the NIC. MAs also connect to one another through a proxy for this reason.

MAs are passed Record objects (via MAPs) from system components. The Records are then dispatched to registered MonitoringAgentListeners based on the type of Records the listeners are subscribed to.

AgentModules

AgentModules are essentially Record queues. They will store an incoming Record for processing by a worker thread. We need this separate worker thread so that processing that takes a long time will not hold up other parts of the monitoring system. (This might happen, for instance, if a module has to wait for a resource.)

The following AgentModules are included in this release:

IPR2025-00200 / 722 Patent Page 11 of 92 EX1010

- FileModule – Writes Records to files. Files are renamed after reaching a given size, and new files are opened each day for any given record. (We use new files each day so that looking up data is easier – files are not deleted, but named such that the filename includes the date of the day they were written). In the first version, log file formats will simply be easily readable. In a subsequent release, log file formats will follow the IPDR standard.
- DbModule – Writes Records to corresponding tables in a database(s) through JDBC. In development, the database will be mySQL. In production, the database will most likely be Oracle on Linux. The database will not affect the implementation, since databases communication is achieved through a 3rd party JDBC driver.
- EmailModule – Emails Records to lucky recipients based on the severity of the Record. Email recipients are subscribed to severity levels, and when a record is received of a given severity, the appropriate recipients are emailed. Likely recipients may include a 3rd party monitoring staff.

The following AgentModules are slated for a post-November release:

- SNMPModule – stores values from received Records in a MIB database, which may be queried through the SNMP protocol by a 3rd party SNMP Manager (NetSaint). The severity property of a received Record can trigger the SNMPModule to send an SNMP Trap to the SNMP Manager.

AppMonitor

The purpose of the AppMonitor is to periodically test the flow of messages through a gateway to all nodes within a given cluster, and to report the results of the test to the monitoring system. Specifications for the tool are stored in the AppMonitor.conf configuration file.

The AppMonitor uses a MonitoringAgentProxy to funnel any messages to the monitoring system. The two primary types of records that the AppMonitor will send to the system include the ExceptionRecord and the AppMonitorRecord.

When the AppMonitor is first launched, it creates an open connection with all nodes in the cluster. If any of the connections fail at any time, an alert will be sent to the monitoring system (using the ExceptionRecord class). At the time of failure, the AppMonitor will immediately attempt to recreate the connection. If the initial reconnect attempt fails, then the AppMonitor will again attempt to recreate the connection the next time that the monitoring test is performed. Each time that the AppMonitor attempts to recreate a connection and fails, another ExceptionRecord will be sent to the monitoring system.

The AppMonitor creates a Timer object that will periodically send prompts back to the AppMonitor indicating that it is time to perform the monitoring test. Each time such a prompt is received, the AppMonitor will broadcast a single time-stamped message to a gateway. Then, it will wait a short time to allow the message to be received at each node

IPR2025-00200 / 722 Patent Page 12 of 92 EX1010

connection. Finally, it will check the message receipt status for each of the node connections. These results will be stored in a histogram (e.g., how many nodes received the connection within 500 milliseconds, how many within 1000 milliseconds, etc.) within an AppMonitorRecord object, which will be forwarded to the monitoring system.

Records

All records have the following fields (as defined by the abstract class *Record*):

- Version
- Type
- Timestamp
- Severity
- String describing the record (implemented by sub classes -- essentially the "toString()" value)

The following Record subclasses and their unique fields are included in this release:

- DebuggingRecord – Used for writing debugging output
  - Class which spawned the record
  - Method which spawned the record
  - String debug message
- ExceptionRecord – used to report a caught or thrown exception
  - Class which spawned the exception
  - Method which spawned the exception
  - The exception caught
  - String debug message
- TBD

SNMP Manager

NetSaint will be the SNMP Manager, collecting SNMP data from the nodes, gateways, routers, and switches at the system level, and utilizing existing SNMP Agents that reside on each of these devices. The data collected will include CPU, memory usage, disk I/O, and uptime.

Fault Tolerance & Fault Detection

IPR2025-00200 / 722 Patent Page 13 of 92 EX1010

MonitoringAgent and *Modules*

These components are created in a Java class. This class will be started by a Perl script. Any failure in the process (any of the modules or the MA itself) will bring all the modules and the MA down. The script will automatically restart the class. (In a future release, Watchdog will also be pinging the MAs. If the component is still running, but unresponsive, then Watchdog will kill the unresponsive script process and start a new one.)

Modules

If a module fails internally and throws an exception, this should cause the entire process (see *MonitoringAgent* and *Modules* section above) to fail and restart. If the module fails without throwing an exception, we will not currently do anything for the following reasons:

1) If we send a Record upstream to the MA, we may cause an infinite loop of messages up to the MA and back to the failed module.
2) If we throw an exception to bring down the process, we may do this infinitely, preventing other non-failed modules from performing their job.

The possible undetected failures are:

1) FileModule cannot write to file (disk full)
2) DbModule cannot connect to one of its databases (network or db down)
3) EmailModule cannot send email through the configured SMTP server

MonitoringAgent Connections

MonitoringAgents have network connections to other MAs and to system components. When an MA connects to another MA, it does so via a MonitoringAgentProxy. When a system component connects to an MA, it does so via a MAP too. The proxy's job is to try to reopen a dropped connection to an MA. While the proxy is trying to reopen the connection, it will use a FileModule to record unsent Records. (Eventually, we may recover and send these Records.)

System Components

If a system component fails this is detected by AppMonitor. It will create an appropriate Record and send it to its MA. If a computer fails, bringing down the system component and MA, both AppMonitor and NetSaint will detect an anomaly.

Cluster Computer Failure

The aggregating computer for a cluster will be redundant. The primary and secondary computer will be identical and watching each other via NetSaint. If one fails, the other handles all traffic and raises an alert via NetSaint.

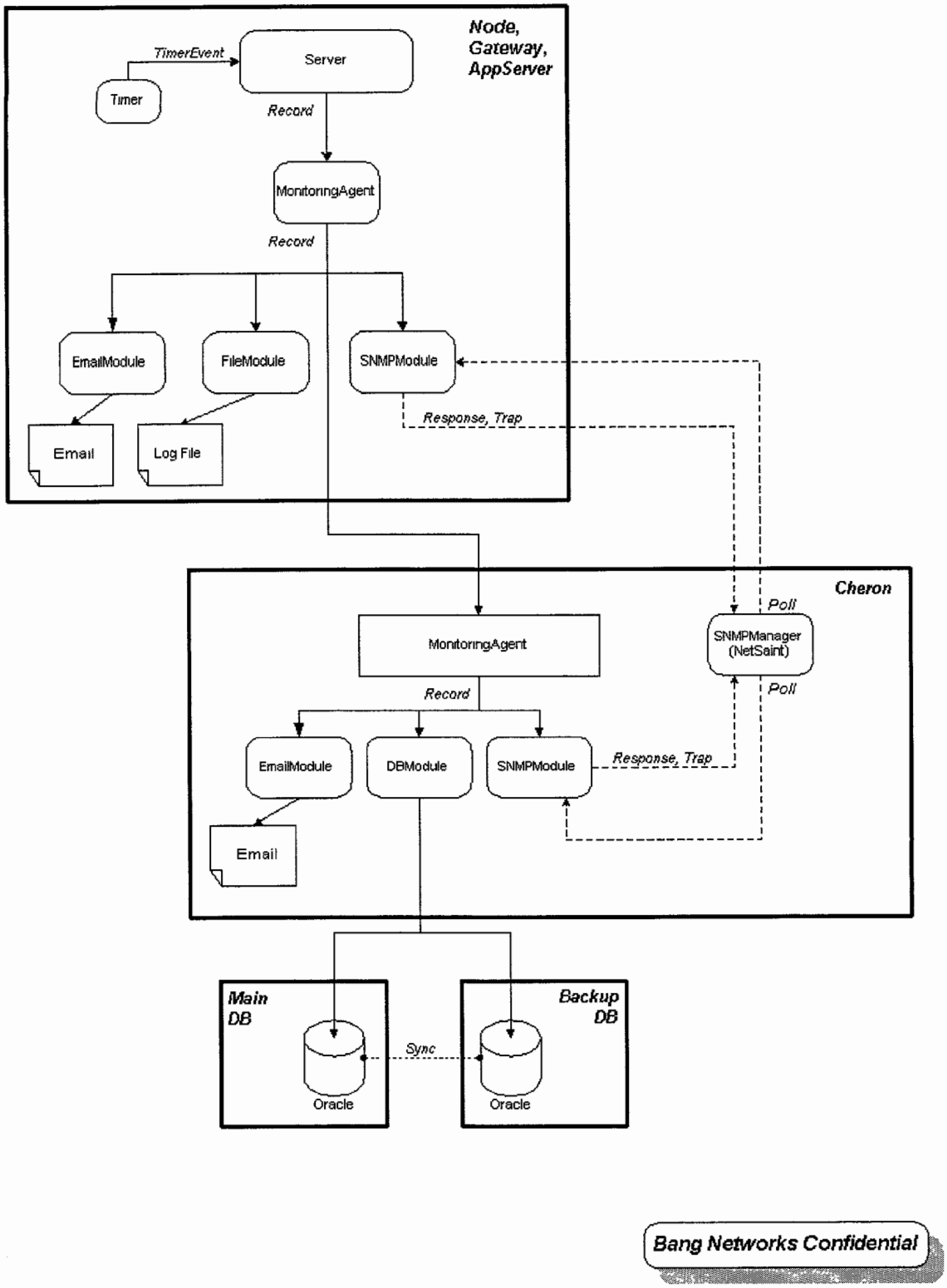
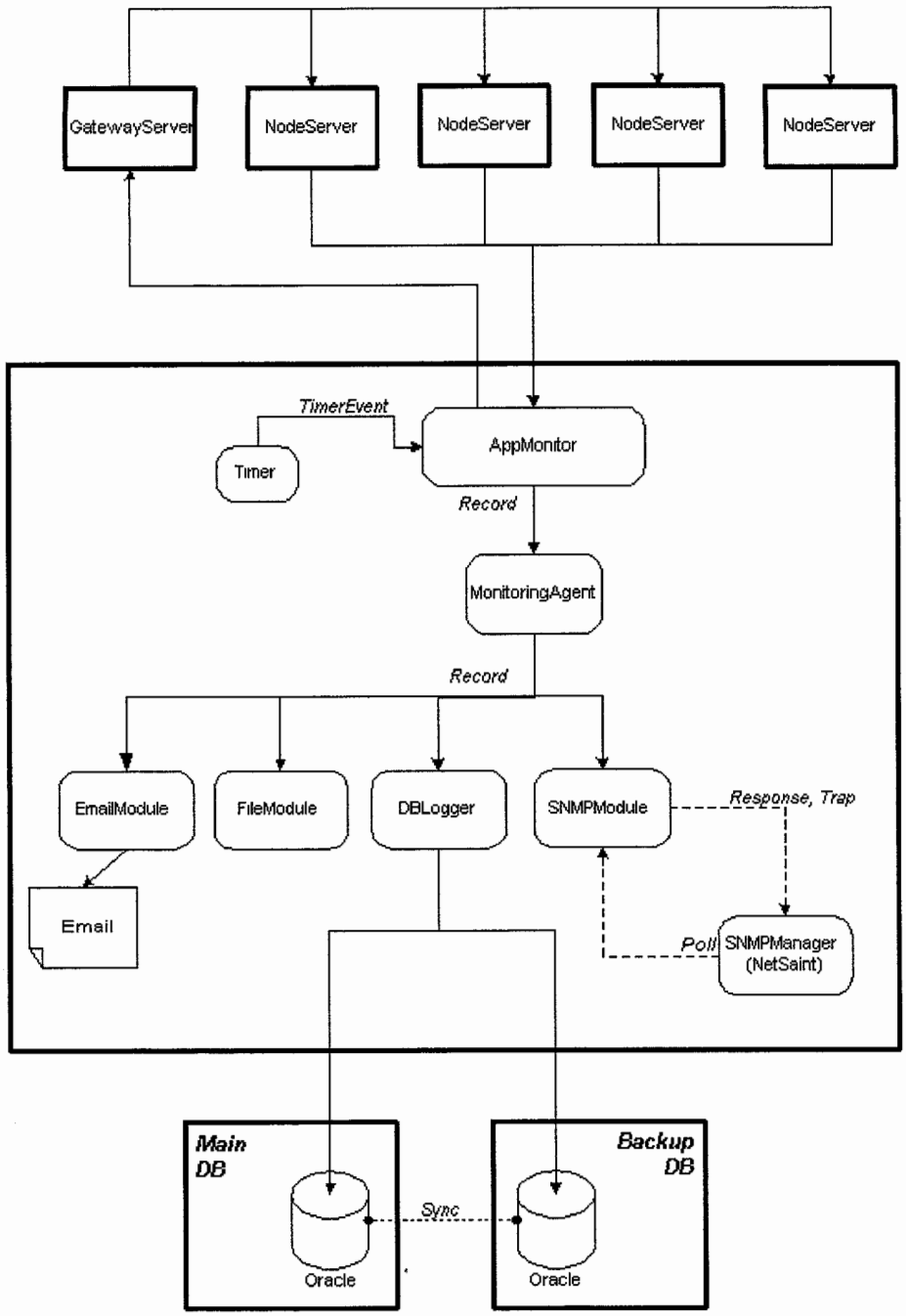


Figure 1: MonitoringAgent Layout, Data Collection



Bang Networks Confidential

MonitoringAgent Layout: AppMonitor

Introduction

The purpose of this document is to specify the goals and proposed solution for monitoring of the Bang Network.

Design Goals

In determining the best monitoring solution for the Bang Network, there are three main goals to achieve. The first is a scalable implementation that will provide the basic monitoring needs in the short run, and can be developed in phases to achieve a fully featured monitoring and logging system. In the short run, system performance such as CPU, disk I/O, and memory usage need to be monitored, and activity that extends beyond established thresholds should trigger alerts. Application uptime and basic application errors should similarly be monitored and trigger alerts.

The second goal is to design a flexible system that conforms to existing monitoring standards such as SNMP. This flexibility allows for integration with multiple third party solutions, such as NetSaint, BMC, and HP OpenView.

The final goal is to keep the system as simple as possible. A simple design is not only easier to develop, but is also easier to verify during QA. Clearly defined modules with specific tasks can be developed and tested in parallel, encourage code reuse, and make maintenance easier.

System Framework

The monitoring framework revolves around a single layerable component called the MonitoringAgent. MonitoringAgents exist as a part of each component in the Bang Network – NodeServers, GatewayServers, etc. There is also a MonitoringAgent associated with each cluster. One MonitoringAgent can dispatch any data it receives to any other MonitoringAgent. For example, the MonitoringAgents on each Node and Gateway propagate the data they receive to the cluster-level MonitoringAgent for aggregation across the cluster.

Each Agent serves as a central dispatcher of Records, which are the individual records of data. Records can be created by any component in the Bang Network, and can represent any type of information, such as debugging info, alert info, or performance info. Each Record is first sent to a single MonitoringAgent, which in turn dispatches it to appropriate modules that handle now what to do with the given Record. These modules have the ability to write to file, database, SNMP, etc. More detail on each module is discussed in a later section.

IPR2025-00200 / 722 Patent Page 17 of 92 EX1010

By funneling Records at any given level (within a Node, within a cluster, etc) through a single MonitoringAgent, we provide a single point of entry for recording Records on any given destination. This alleviates issues associated with resource contention when trying to write multiple items to disk, database, etc. If each Node had its own ability to write to a cluster-level database, there would be contention issues when multiple Nodes tried to write records to the database at the same time. The single entry point provided by the MonitoringAgent design eliminates this issue, and simplifies the task of handling I/O to a single resource (file, database) from multiple sources.

The layered approach meets the goals of scalability, and simplicity of design and implementation. The task of collecting Records and handling them appropriately happens at many different levels: within Nodes and Gateways, within a cluster, and system wide. Each component has well-defined and independent set of tasks, so development can occur in parallel.

Timer

The Timer is the simplest component in the monitoring system. This utility is configured to trigger TimerEvents on a given time interval, which in turn may invoke methods on one or more components. For example, a Node may create a Timer set to trigger a TimerEvent every 5 minutes, starting at 12:00am. At 12:00am, the Timer would activate the TimerEvent, which is set to calls a method on the Node. That method could dump internal state counters to a MonitoringAgent in the form of Records, which in turn may be written to file, database, etc. In this way, the Timer is used throughout the monitoring system to trigger data collection at fixed times and intervals*.

** Note: It is assumed that all boxes will have their system clocks synchronized via a tool such as ntp. This ensures, with an acceptable level of accuracy, that all timestamps and TimerEvents occur at the same time on each box.*

Records

Records are the containers of monitoring and logging data. Each Record represents a single entry or data point, such as a debug statement, an application exception, or a time slice of customer usage within a node. Records have categories associated with them, which define their destined location. For example, a debugging Record may only be destined for a log file, whereas customer-billing data may be destined for a log file, a database, and any other MonitoringAgents (so that it may be forwarded through multiple layers of the monitoring framework). Records are created anytime some piece of data needs to be recorded or acknowledged by the monitoring system. Once the Record has been created, it is handed off to a MonitoringAgent and forgotten. It is the job of the MonitoringAgent to handle the Record from that point forward. In this way, the performance of the Node process (or whatever software component that generated the Record may be) is not affected by the tasks associated with handling the Record, such as disk I/O or database transactions. It simply sends and forgets the Record.

The following is a break down of the proposed set of Records:

DebuggingRecord

It is extremely attractive to incorporate debugging information in the same framework as monitoring and logging for the sake of consistency. Debug statements are used thoroughly in development, and using the same methods for debugging as logging and monitoring encourages consistent development coding style.

Each debugging Record will contain a Timestamp, the class that spawned it, the method within the class where it was created, and a String field for specific debugging messages. Debugging Records will only be written to file(s) locally.

ExceptionRecord

Application Exceptions are any number of application level errors that are caught within the code, but should be reported to file and/or the monitoring system to trigger an alert. Each record will contain a timestamp when the error occurred, the class and method in which it occurred, a severity level (most likely 3) which may be used by the monitoring system to determine the type of alert to send, a stacktrace, and an extra field for any other potentially useful information that could be used in debugging the exception. Application Exceptions will be written to file, written to db, passed to the SNMPModule (discussed later), and passed to any other listening MonitoringAgents for aggregation.

CustomerBillingRecord

Billing data are Records that contain the basic points of interest per customer that would be used in billing. These Records will be generated both at the node and gateway at a fixed interval (probably every 5 minutes), and will contain a Timestamp, the customer ID, number of current connections, and the sum total number bytes sent for that customer since the last time interval (including an estimated byte value for headers per message). It is important to track customer traffic out of both gateways and nodes, so that we're sure to bill customers broadcasting tons of data into our gateways that have no client connections at the nodes. These Records will be written to file, written to db, and passed to any other listening MonitoringAgents for aggregation.

Histogram Data

There are any number of data points that we would like to collect which reflect usage and statistics of the network. This data isn't currently used for billing, but is particularly interesting as we begin to understand usage patterns of clients and customers. While it is infeasible to store a record of every transaction across our network, histograms can reflect general statistics. Histograms we would like to collect are the duration of connections per customer, the size of messages sent per customer, and the frequency of messages sent

IPR2025-00200 / 722 Patent Page 19 of 92 EX1010

per customer. These histograms are recorded in a Record with customer ID and timestamp. Histogram data will be written to file, written to db, and passed on to any other listening MonitoringAgents for aggregation.

Performance Monitoring

TBD

MonitoringAgent

The MonitoringAgent is the central component in the monitoring framework, and plays the role of Dispatcher for incoming Records. Each MonitoringAgent will maintain a set of registered listeners. When a MonitoringAgent receives a Record, it asks each listener if it subscribes to this type of Record, and if so, forwards it to the listener. (Alternatively, the MonitoringAgent could ask for subscription information at registration and then only forward Records based on that subscription information. Unfortunately, this does not allow the listeners to change their subscription information at runtime.)

MonitoringAgents will run locally on each software component of the Bang Network, such as the Node, Gateway, and Application servers. There will also be a single MonitoringAgent running per cluster, which will be a registered listener to the MonitoringAgents on each Node and Gateway in the cluster. In this way, Records received within each Node can be quickly passed off and aggregated in the database at the cluster level. See attached diagrams for details on MonitoringAgent layout.

MonitoringAgent Modules

As mentioned earlier, MonitoringAgents dispatch Records to various modules, one of which may be another MonitoringAgent. These modules encapsulate the ability to queue and then transfer a Record to a given type of media or communication protocol. The following modules are forecasted for use with the November release, and should be sufficient in the long run.

FileModule

The FileModule has the knowledge of taking a Record and writing it out to an appropriate file on disk. In general, each type of Record will have its own log file. Log files will be in ASCII format, and the format of each record in the file will most likely follow the Internet Protocol Detail Record (IPDR) format, or a similar alternative (The IPDR is a standard log file format proposed by Narus and supported by a number of other companies). The FileModule is also responsible for aging files on a given interval (daily, weekly, monthly), renaming aged files, and opening new files as needed. However, it will not delete files, but simply keep them organized so that data from them can easily be located (or aggregation at a cluster can be more easily recreated). Deletion/archiving of files will be handled by another utility to be introduced at a later date.

DatabaseModule

The DatabaseModule, like the FileModule, knows how to take a Record and write it into an appropriate table in a database.

EmailModule

The EmailModule receives Records and, based on their severity, emails out their contents as an immediate alert. This functionality is intended to provide a short-term solution to the need of application level monitoring and alerting. These emails will most likely go to a pager or central console monitored 24/7. In the long term, all alerts and email will be dispatched by a single entity (most likely the SNMP Manager/3rd party software).

SNMPModule

The SNMPModule serves as the application level hook between the MonitoringAgent and a third party SNMP Manager (described below) such as HP OpenView, NetSaint, or BMC. This module will not be implemented in the short run due to time constraints, but is an important component of the overall system, and should be in place in time for a full product launch.

The Bang SNMPModule is both proactive and reactive. When Records are passed to the SNMPModule from a MonitoringAgent, it checks for a property on the Record that tells the agent whether or not to behave proactively. Reactive Records will commonly contain the current state and latency values of the network, and the SNMPModule should simply hold the Record in a small internal cache. When the SNMP Manager polls the SNMPModule for data, the SNMPModule 'reacts' by replying with the values stored in this cache. Should the Record property be set to proactive, as soon as the Record is received, the agent should immediately send an SNMP Trap. Traps are SNMP messages actively sent to the SNMP Manager to indicate an unusual state, which in turn may trigger the SNMP manager to send an alert. Proactive Records are important data points that should be reported immediately, such as notification that a node has gone down, a node cache has been purged, etc.

SNMP Manager

The SNMP Manager is a third party tool such as NetSaint, HP OpenView, or BMC. In steady state, SNMP Managers poll SNMPModules to collect status data. They may also listen for SNMP traps actively sent from an SNMPModule. Collected data may be any specific value of interest. At the system level, it might be CPU levels, memory utilization, or free disk space. At the application level (as collected by the SNMPModule described above), the data might be the latency of message delivery at each node. The SNMPManager is configured with threshold levels for each type of data it collects. As this data is collected, should the data value exceed the configured threshold, the SNMPManager triggers alerts. These alerts may take the form of a blinking light on a visual console, an email to a trouble ticket system, or a page to some lucky engineer.

IPR2025-00200 / 722 Patent Page 21 of 92 EX1010

The SNMPManager may also receive SNMP traps from SNMPModules, which are messages that tell the Manager something important has happened, and immediate attention is needed. Traps are generally only triggered for serious problems such as 100% CPU utilization or the loss of connectivity to a node.

AppMonitor

Performance and application monitoring is achieved with the AppMonitor. The AppMonitor is an adaptation of the Benchmark Tool. Each colo will house a single AppMonitor, which holds open connections to each node in the cluster. A timer is associated with the AppMonitor, and at each tick of the timer, the AppMonitor delivers a timestamped BMPMessage to a GatewayServer and records the system time when the message is received from each Node. The difference of these timestamps measures the latency of the Bang Network for each node, and is recorded in a Record. Should any node connection be lost, the AppMonitor recognizes this as an event and delivers a Record to its local MonitoringAgent to be handled appropriately (see the SNMPModule description above). In this way, both the performance level and the "uptime" of nodes can be determined in each cluster.

Cluster Database

Each cluster will house a database that handles the collection and aggregation of data across the nodes and gateways of a cluster. The database receives data via the DBLogger module. This data can then be queried for cluster-wide statistics for billing and usage.

Fault Tolerance
The cluster database will be a relatively low-cost solution, with a single redundant back up. The DBLogger will most likely send all Records to both the main database and the backup database. In the case of a failure of one database, the second will continue to receive and record records. Once the failed DB has recovered, the two databases can be synched.

Future Tasks

There are still several pieces not addressed in this document, which are required for a robust, long-term monitoring system. These include, but are not limited to:

- File aging and archiving on each box. Old log files must be archived or removed periodically so as not to overwhelm the storage space on any device. In the short run, this task can be done manually, but a more robust cron task or other automated system must be in place in the long term.
- Monitoring of the MonitoringAgents. It is important to put an additional layer of checking on each process, which monitors our agents and logging components. This could be achieved via perl scripts, which periodically ping the

IPR2025-00200 / 722 Patent Page 22 of 92 EX1010

MonitoringAgents and Timers on each box. This would verify the responsiveness of MAs and give us an opportunity to kill an unresponsive MA and start a new one.

- Monitoring of the cluster database. This may be solved through 3rd party monitoring support, but nonetheless must be addressed soon.

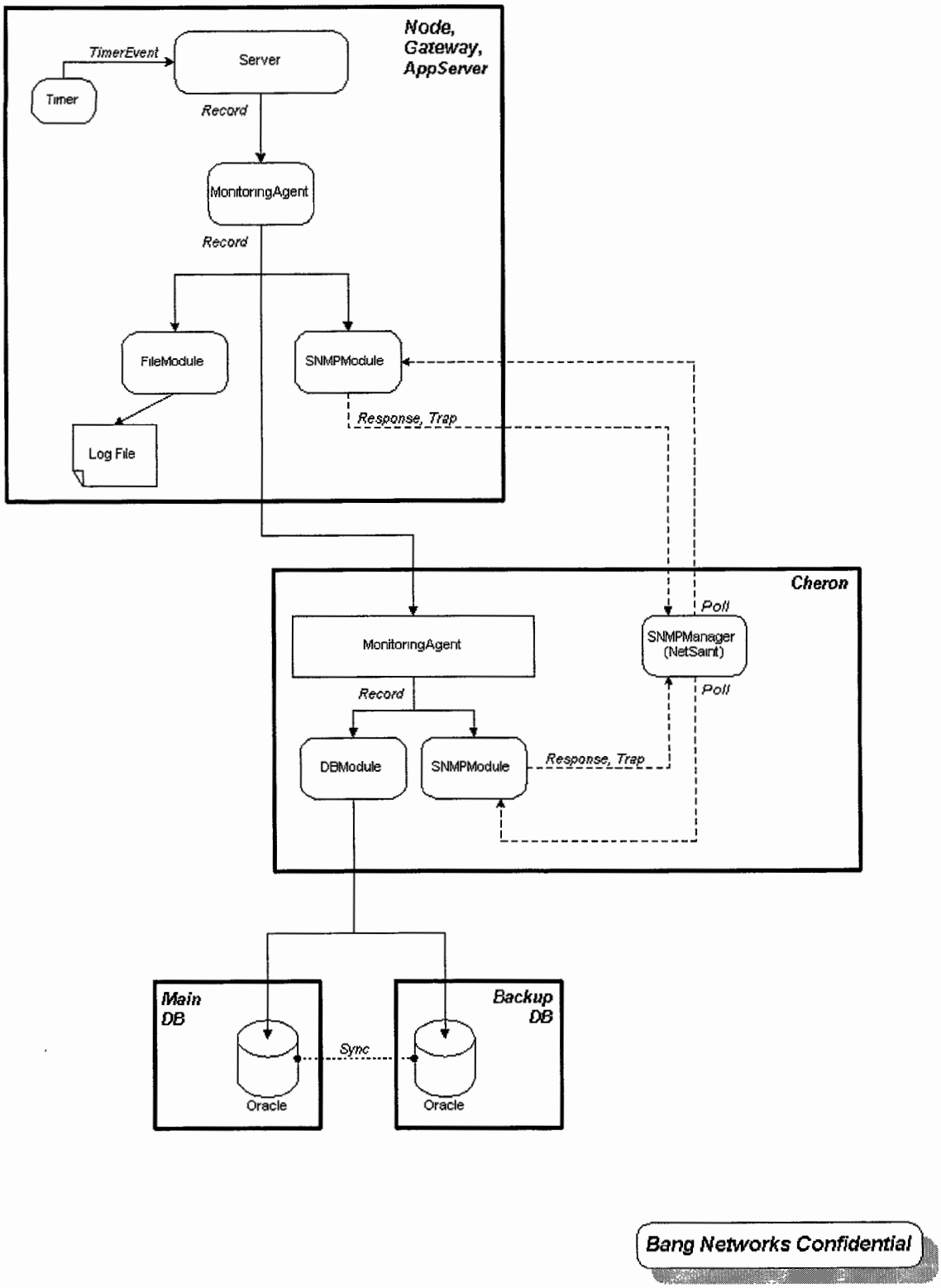
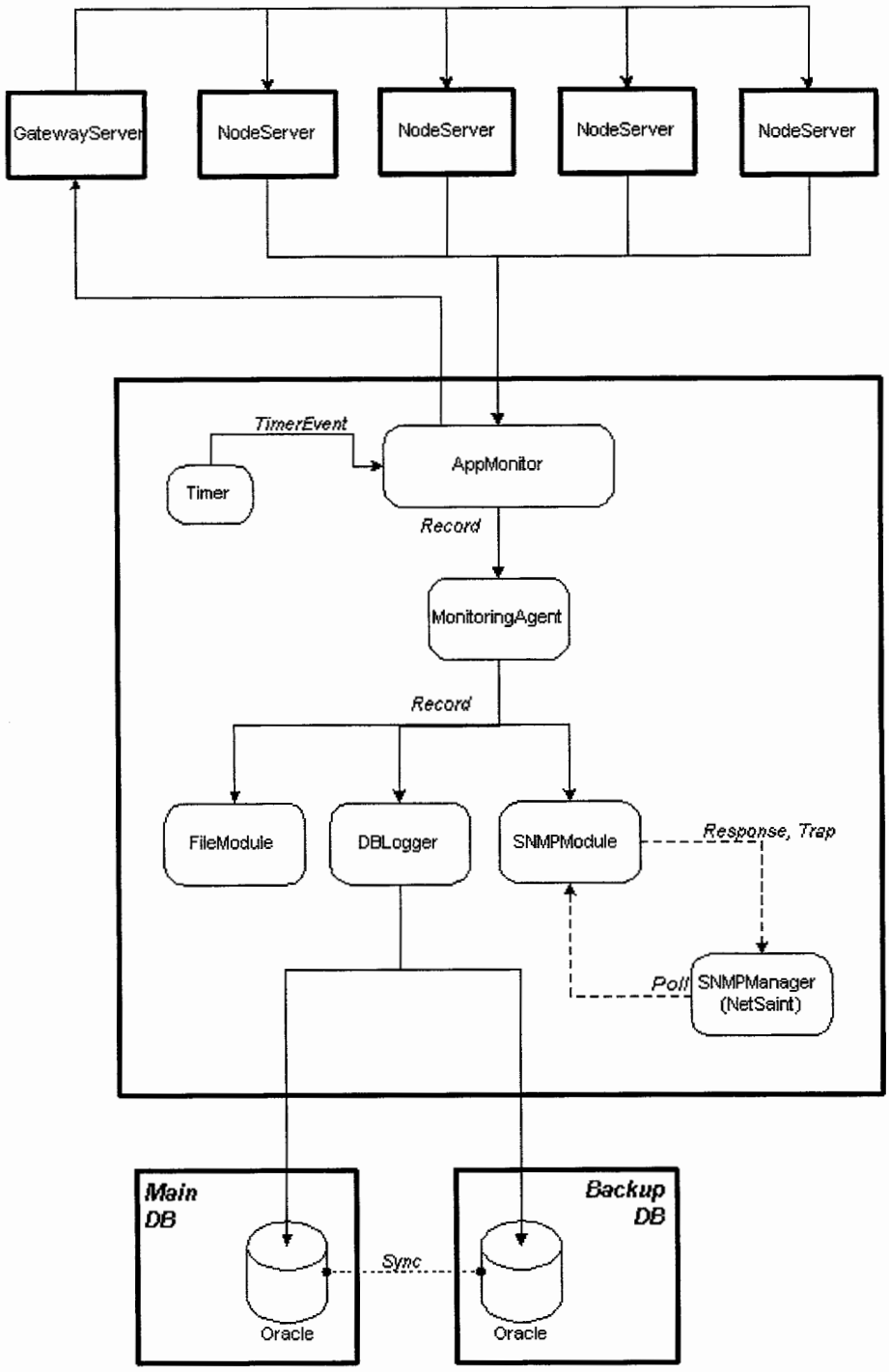


Figure 1: MonitoringAgent Layout, Data Collection



Bang Networks Confidential

MonitoringAgent Layout: AppMonitor

Specification for implementing object differentiation ("diff'ing") into the DataPump

Introduction

On successive reads of an XML file, the DataPump cannot differentiate between data that has changed, and data that is the same as the previous read. As a result, the DataPump since each message in the file, regardless of whether it has actually changed. This results in a significant amount of unnecessary network traffic.

Interfaces

DataPump
BangAPI – No changes anticipated.
Gateway – No changes anticipated.

Functionality

The DataPump will maintain a record, in memory, of objects it has previously sent, and will compare the value previously sent, with the most recently read value. If the value is different, the new data will be distributed over the network, otherwise, the message will not be sent.

To provide a modicum of control over memory usage, the revised specification for the Bang XML (see datapump-config.dtd) files will allow the customer to specify how many messages should be kept in memory. If this threshold is exceeded, the earliest stored messages will be discarded.

To provide a degree of control over the performance impact of diff'ing, and to allow data to be sent regardless of whether it has change, the customer will be able to specify whether objects should be diff'ed at both the file, and individual object level. This setting will be controlled through an attribute in a revised specification of the Bang XML files (see bang-message-collection.dtd).

Implementation

Modify the XML parsing in the DataPump to check for the new compareToLast attributes, and act appropriately.

The actual "diff'ing" functionality will be implemented using two data structures from the java.util package. A java.util.HashMap will be implemented to store previously parsed objects. The key value for the HashMap will be formed by concatenating the value of the objectID attribute in the <bang-message> element, with "$" and the value of the name attribute in the <property> element. Additonally, a java.util.LinkedList will be implemented as a FIFO structure (queue) and will contain all of the keys currently stored in the HashMap. When the size of the of the HashMap reaches the specified limit, this structure will be used to determine which key-value pair should be removed to make room for new data.

The proposed implementation will share the diff'ing functionality across multiple XML files, and thus, care will need to be taken to make the implementation thread safe.

Test Plan

A number of XML files will need to be created to test that diff'ing is functioning properly. Initial testing will focus on comparing the output of the data pump logging with expected behavior. A more thorough test would involve writing a custom listener object to ensure that messages weren't being sent unwittingly, but this is outside the scope of current time constraints.

Functional Specification for Gateway Fault Tolerance

Introduction

In order to bootstrap the system, first Gateway servers must be started. Then, all node servers connect to the existing Gateways, known through the NodeServer.conf files. However, there is no notification mechanism in the system to indicate that a new Gateway has been introduced after all the nodes have been bootstrapped.

The system needs to be able to gracefully handle new Gateways that are introduced into the network.

Interfaces

Gateway servers can talk to another load balancer-selected Gateway server through the by sending broadcasts through the load balancer (see GatewayServer.sendMessageToGateway()). Node servers connect directly to Gateway servers within the Bang network and maintain those persistent connections.

Functionality

New Gateway: When a new Gateway *A* comes up, a message should be sent to the Bang network so that all the nodes will connect to the new Gateway. If connections can't be established with all the nodes, a trouble ticket should be logged.

Node Loses Connection with Gateway: For the scenario where a NodeServer loses connection with a Gateway, the NodeServer should try to reconnect. If, after a specified interval, the NodeServer can't reconnect, log a trouble ticket.

Implementation

New Gateway: When a new Gateway *A* starts, it announces its existence to the system by sending a GWAlive to the load balancer. There may not be another Gateway on the other side if this is the first Gateway in the system to be started.

If there are live Gateways already, they will be listening through the apiListenPort (definable in Gateway.conf), and the load balancer will choose a Gateway *B* to receive the message. The following sequence of events should then occur:

1. Upon receipt of this GWAlive message, *B* will return to *A* the number *n* of nodes that *A* should be connected to, through an AuthenticatedGenericMessage.
2. Upon receipt of *n*, Gateway *A* sets up its listening port (definable in Gateway.conf as nodeListenPort) for nodes.
3. Gateway *A* schedules a task NodeConnectionsTimerTask through a timer that waits *t* milliseconds (configurable through Gateway.conf's allNodeConnectedTimeLimitMillis), which limits the amount of time it waits to get *n* connections.
4. Gateway *A* sends a GWAliveStartNodeConnections message, which contains Gateway *A*'s host, port, and ID.

5. Upon getting the GWAliveStartNodeConnections message, Gateway *B* then generates a GWAlive message that it sends to the nodes.
6. Gateway *B* then sends the GWAlive message (with the relevant information from GWAliveStartNodeConnections) to the nodes that it is connected to. The nodes grab the Gateway *A*'s information from the received GWAlive message, and attempt to connect to it.
7. Meanwhile, Gateway *A* waits until it establishes connections with *n* nodes. If *t* elapses before all the nodes establish connections with *A*, then *A* closes any open connections by sending ConnectionClose, and tries the process again, starting with sending a GWAlive message to the load balancer.
8. When Gateway *A* gets all *n* connections to all the nodes, it then makes itself available to client connections that connect through the API by opening up the designated apiListenPort. If *A* fails to ever successfully get connections from all the nodes, it exits, logging a trouble ticket.

Node Loses Connection with Gateway: If a node loses connection with a Gateway, the node should try a number of times to reconnect. If it fails, the node logs a trouble ticket and exits.

Logging: The servers connect to the MonitoringAgent process to log messages.

Changes to Configuration File: What follows are segments from the Gateway.conf dtd, where changes have been made in **bold**.

```
<!ELEMENT GatewayServer (gatewayPort?, gatewayHost?,
GatewayServer, Security, Logger?, Debug, ConnectionManager)>

<!-- gatewayPort and gatewayHost tell the gateway how to get
into the network through the load balancer. This way, a new
gateway can talk to another gateway and not have to know
particulars about other gateways' host/port information. -->
<!ELEMENT gatewayPort (#PCDATA)>

<!ELEMENT gatewayHost (#PCDATA)>

<!ELEMENT GatewayServer (apiListenPort, nodeListenPort,
initNumTries?, initWaitTimeMillis?,
allNodesConnectedTimeLimitMillis?)>

<!-- This is the port that gets registered with the load
balancer, and it's the port where messages from broadcast
clients ultimately end up. This port is opened after the
gateway has finished its initialization. -->
<!ELEMENT apiListenPort (#PCDATA)>

<!-- This is the port through which gateways and nodes
communicate. It is opened during the initialization phase of
the gateway. -->
<!ELEMENT nodeListenPort (#PCDATA)>

<!-- The amount of time to wait before the gateway gives up
trying to connect to the load balancer during initialization.
It does this step when trying to figure out if there are other
gateways already alive. Defaults to 1000 ms.-->
<!ELEMENT initConnectTimeoutMillis (#PCDATA)>
```

```
<!-- The number of times to try establishing connections with
all the nodes  (in the case where we're a new gateway coming
into a system where there are nodes and gateways already up)
Defaults to 5. -->
<!ELEMENT initNumTries (#PCDATA)>

<!-- The amount of time to wait between tries to initialize all
nodes (in the case where we're a new gateway coming into a
system where there are nodes and gateways already up). Defaults
to 1000 ms. -->
<!ELEMENT initWaitTimeMillis (#PCDATA)>

<!-- The amount of time to give the gateway to fulfill
connections with all expected nodes (in the case where we're a
new gateway coming into a system where there are nodes and
gateways already up). Defaults to 10000 ms. -->
<!ELEMENT allNodesConnectedTimeLimitMillis (#PCDATA)>

<!—Logger stuff here →
<!ELEMENT Logger (loggingPort)>

<!—the port of the monitor agent logging process->
<!ELEMENT loggingPort (#PCDATA)>
.
.
.
```

Security

All the messages involved in the Gateway Fault Tolerance system must be authenticated. Thus, all existing and gateways and nodes must have the id and key of a new gateway that is introduced into the system. See Outstanding Issues regarding possible issues with this requirement.

Outstanding Issues

Key distribution: Because all of the messages involved in the system are authenticated, the gateways and nodes must know the ID and key of the new gateway. This requirement can make key distribution difficult for new gateways. A proposed solution is to have the ID/key pairs in a central database. As a stopgap solution, the NodeServer and GatewayServer will re-read their respective .conf files to get any new keys. We may want to consider creating a tool to automatically add a new ID/key pair to configuration files.

Load balancer configuration: Introducing new gateways to the load balancer is currently a manual process, which requires interaction with a web UI. This may not matter in the short run, considering the infrequent occasions that brand new gateways will be introduced into the system. In the long run, it may behoove us to at least create command-line scripts to automatically add a gateway into the gateway pool.

Integration with monitoring system: It's also not yet clear how a monitoring system will integrate into the Gateway fault tolerance mechanism.

Test Plan

IPR2025-00200 / 722 Patent Page 30 of 92 EX1010

Test Environment

The testing environment consists of pre-made scripts running on tgate and tnode. In the CVS hierarchy, there's a bin/test directory with the following structure:

```
bang/
  bin/
    test/
      gatewayFaultTolerance/
        gatewayServer1
        gatewayServer2
        gatewayServer3
        nodeServer1
        nodeServer2
```

These scripts point their respective servers to .conf files the appropriate places under bang/conf/test/gatewayFaultTolerance by setting the JAVA_DEV_ROOT environment variable.

The basic configuration for running the GatewayServer scripts is as follows:

GatewayServer Script	Machine	apiListenPort	NodeListenPort
gatewayServer1	tgate	12921	22921
gatewayServer2	tgate	02921	42921
gatewayServer3	tnode0	12921	22921

For NodeServer scripts:

NodeServer Script	Machine	Port
nodeServer1	tnode0	32921
nodeServer2	tgate	32921

Similarly, there are scripts that start up the MonitoringAgent process, which accepts log messages from the servers. The logging ports are defined in the appropriate GatewayServer.conf or NodeServer.conf file, to match the port defined in the appropriate Monitoring.conf file. There's a script to start up a monitoring agent process for each server process.

Script	Machine	Logging Port
GatewayServer1MA	Tgate	12341
GatewayServer2MA	Tgate	12342
GatewayServer3MA	Tnode0	12343
NodeServer1MA	Tnode0	12344
NodeServer2MA	Tgate	12345

For testing, the F5 load balancer has been configured with a virtual server at 172.16.2.212:12921. The virtual server points to the pool GatewayFaultToleranceGW, which consists of the following:

tgate: 10.10.10.5:12921

tgate: 10.10.10.5:12921
tnode0: 10.10.10.4:2921

For the test driver programs to run correctly, the bang/conf/api.conf file must point to 172.16.2.212:12921, and bang/conf/AuthenticationManager.conf must have myID set to CustomerId and myKey set to CustomerKey.

Test Cases

Test	Description	Comments
1	Unit-test new GWAlive, GWAliveStartNodeConnections, and AuthenticatedGenericMessages	(Test case 3 validates GWAlive, GWAliveStartNodeConnections, and AuthenticatedGenericMessages).
2	Bootstrap 2 Gateways and 2 NodeServers, and run test suite, to ensure normal operation.	1. Start gatewayServer1 and gatewayServer2 on their respective machines, according to the GatewayServer Script table above. 2. Run nodeServer1 and nodeServer2 on their respective machines, according to the NodeServer Script table above. 3. Run com.bangnetworks.tests.TestBroadcastManager.
3	Bootstrap 2 Gateways and 2 NodeServers. Bring up 3rd Gateway. System should be properly notified so that nodes establish connections with Gateway.	1. Start gatewayServer1 and gatewayServer2 on their respective machines, according to the table above. 2. Run nodeServer1 and nodeServer2 on their respective machines, according to the NodeServer Script table above. 3. Start gatewayServer3 on tnode0. 4. Run com.bangnetworks.tests.TestBroadcastManager.
4	Bootstrap 2 Gateways and 2 NodeServers. Bring up 3rd Gateway, but simulate the Gateway's inability to receive the expected number of node connections. The new Gateway should retry the whole process in order to re-establish connections with all the NodeServers. If the Gateway doesn't make it through the initialization phase, log a trouble	1. Start gatewayServer1 and gatewayServer2 on their respective machines, according to the table above. 2. Modify conf file for nodeServer1 so that it the id/key pair for gatewayServer3 is commented out. 3. Run nodeServer1 and nodeServer2 on their respective machines, according to the NodeServer Script table above. 4. Start gatewayServer3 on tnode0. Its startup should fail. See log file in conf/test/gatewayFaultTolerance/gateway3.

	ticket.	5. Run com.bangnetworks.tests.TestBroadcastManager. The rest of the system should run as usual.
5	With 2 Gateways up, permanently bring down a Gateway. Ensure Node logs trouble ticket after losing contact with Gateway	1. Start gatewayServer1 and gatewayServer2 on their respective machines, according to the table above. 2. Run nodeServer1 and nodeServer2 on their respective machines, according to the NodeServer Script table above. 3. Bring down gatewayServer2. Nodes should properly log the loss of contact to gatewayServer2. See log files in conf/test/gatewayFaultTolerance/nodeServer1 and conf/test/gatewayFaultTolerance/nodeServer2
6	System should deal with brand new gateway, without prior knowledge of it's key.	1. Modify conf files for gatewayServer1 and gatewayServer2 so that the key for gatewayServer3 is commented out. 2. Modify the conf files for nodeServer1 and nodeServer2 so that the key for gatewayServer3 is commented out. 3. Start gatewayServer1 and gatewayServer2 on their respective machines, according to the table above. 4. Run nodeServer1 and nodeServer2 on their respective machines, according to the NodeServer Script table above. 5. Start up gatewayServer3. It should fail. 6. Modify the conf files to include the id/key pair for gatewayServer3. 7. Start up gatewayServer3.

Network Safeguards

Technical Specification

Introduction

This document presents the technical specifications for the network safeguard components of the Bang Network that are to be included within the beta release of late November. Functional descriptions of the components can be found in the Network Safeguards Product Requirements Document.

The following technical components are targeted for the November release:

a) At the gateway (related to message throughput):
   - The message queue data structure
   - The message purging mechanism
   - Interactions with the monitoring agent

b) At the node (related to message throughput):
   - The message queue data structure
   - The message purging mechanism
   - Interactions with the monitoring agent

c) At the node (related to connections):
   - The connection count data fields
   - The connection refusal mechanism
   - Interactions with the monitoring agent

Gateway safeguard components for message throughput

The message queue:

The purpose of the message queue is to serve as an interim storage place in which to hold messages after they have been received from the customer and before they have been sent to the nodes. Messages within the queue should be processed in first-in-first-out (FIFO) order. The record for each message should include the following:

- The customer that sent the message.
- The time that the message was received.
- The size of the message (in bytes).
- The message itself.

In addition to the message queue itself, the node should a single variable that holds the current number of messages in the queue, as well as an array that represents a histogram

of the number of times that the message queue has been at different lengths over some recent interval (actually, the length of this interval will correspond to the polling interval at which this information is sent to the monitoring agent).

The method for managing the histogram of queue length statistics should be as follows. First, each time a new message is added to the queue, add one to the variable that stores the current length of the queue. Then increment the corresponding bucket within the histogram. For example, the code might look like this:

```
currentLength++;
queueLengths[currentLength]++;
// which could be shown on one line as queueLengths[++currentLength]++;
```

Similarly, each time that a message is removed from the queue, decrement the current queue length counter and then increment the corresponding bucket within the histogram. For example:

```
currentLength--;
queueLengths[currentLength]++;
// which could be shown on one line as queueLengths[--currentLength]++;
```

The message purging mechanism:

The purpose of the message purging mechanism is make sure that the gateway does not become overloaded with messages. The purge mechanism, which will be triggered when the size of the message queue grows too large, involves discarding a subset of messages from the queue without sending these out to the nodes (in other words, the discarded messages are lost to the system).

In order to implement the queue purging mechanism, two cutoff levels must be established:

- The "purge" level
- The "rollback" level

Of these, the purge level represents the maximum number of messages that can be stored within the queue before a purge is triggered. The rollback level indicates the maximum number of messages to leave in the queue after the purge.

The general algorithm for testing whether or not to purge, and if necessary performing the purge, can be represented as follows:

```
// Assume that a message has just been added to the queue
// As a result, the counter representing the length of the queue has been incremented
if (currentLength >= PURGE_LEVEL) {
    // Trigger alert to monitoring agent
```

```
        alertPurge();
        // Purge records from the end of the queue
        while (currentLength > ROLLBACK_LEVEL) {
            // Pop the most recent message – let it be discarded
            messageQueue.getLast();
            currentLength--;
        }
    }
```

Interactions with the monitoring agent:

The safeguards related to message throughput at the gateway will interact with the monitoring system in two important ways. First, at regular intervals the current state of the queue length histogram should be reported to the monitoring system. Second, whenever the message purge mechanism is invoked, an alert message should be immediately dispatched to the monitoring system.

Note that in each of these cases, the information should be targeted for all possible output locations (i.e., for file storage, for downstream monitoring agent processing, and for SNMP/NetSaint processing).

In order to facilitate such interactions, the safeguard code must have access to the following:

- A variable that stores the polling interval
- A Timer object to prompt for messages to be sent at the polling interval
- A reference to the MonitoringAgent object used at the gateway level, to which the messages can be sent.

Node safeguard components for message throughput

The message queue:

The message queue within the node should be for all intents and purposes identical to that within the gateway. Similarly, the node should also track the current number of messages in the queue, as well as a histogram for the number of times that the queue has been at different lengths over a specified interval.

The message purging mechanism:

The node will also use the same purging mechanism as does the gateway. However, given the fact node may need to transmit a message to a larger number of connections than would a gateway, it may be appropriate to specify a different "purge" level and "rollback" level.

Interactions with the monitoring agent:

The message throughput safeguards at the node will need to implement the same types of communication with the monitoring system as the gateway: specifically, to send at specified intervals the current state of the queue length histogram, as well as to send an immediate alert whenever a message purge occurs.

In order to support this interaction with the monitoring system, the node (like the gateway) will need a variable to store the polling interval, a Timer object to prompt for messages to be sent according to the polling interval, and a reference to the MonitoringAgent object used at the node level.

Node safeguard components for connections

The connection count:

Here, it is simply necessary to keep an accurate count of the total number of active connections at the node. Obviously, this counter should be incremented each time a new connection is accepted, and should be decremented each time a connection is lost or dropped.

The connection refusal mechanism:

If the node accepts too many connections, its performance in terms of message throughput rates may grind to a crunching halt. For this reason, it is necessary to ensure that the number of open connections is not allowed to grow over some specified upper limit (the value of which, incidentally, should be determined as a result of the stress testing efforts). The mechanism for this is simply to check the current number of active connections when any new connection attempt is made. If the current number is less than the specified maximum, then it is deemed appropriate to accept the connection; otherwise, the new connection is refused.

Interactions with the monitoring agent:

At some fixed interval it is necessary to send the current number of active connections at the node to the monitoring system. In order to make this happen, it is necessary to include:

- A variable that stores the polling interval for the total connections information
- A Timer object to prompt for messages to be sent at the polling interval
- A reference to the MonitoringAgent object used for the node, to which the messages can be sent

Network Safeguards

Possible Implementation Strategies

Introduction

The purpose of this document is threefold: first, to identify possible problems related to excessive message throughput and user connectivity that could potentially overwhelm, and temporarily cripple, the Bang Network; second, to introduce a variety of potential strategies that may be implemented in order to prevent such occurrences; and third, to serve as a reference point for discussions between the product development and the engineering teams, who together must decide the most appropriate strategies to pursue.

Once such decisions have been made, this document will provide the basis for a technical specification for the implementation of network safeguards.

Guiding Principles

The technical issues surrounding network safeguards are quite challenging and complex, particularly with respect to aggregate system behavior. As a result, there are in many cases no obviously "right" answers. For example, one solution may guarantee that the system will not fail, but do so at the price of dropping messages or client connections indiscriminately. Another solution may seek to ensure that all customers receive "fair" treatment (with respect to their contracted levels of service), but in the process introduce certain strategies which, once scaled across the entire network, lead to unpredictable or chaotic behavior that could cause the network to fail.

Given these considerations, it is useful to establish a set of guiding principles that can help in the process of evaluating and selecting different strategies to implement. Toward this end, the following list of guidelines is introduced. Note that the list represents the author's best understanding of the collective opinions of product development and engineering. There are a number of items, however, that are likely to prove controversial none-the-less. As such, the list should certainly be considered as a proposal around which to base discussion rather than as the final word.

The principles outlined to date are as follows:

1) The overriding goal must be to ensure that the network does not fail.

2) The secondary goal (i.e., subject to the primary goal) is to make sure that the service for *non-offending* customers (customers whose level of system utilization falls within their contracted levels) is not impacted by the actions of *offending* customers (customers with excessive client connections or with excessive size and/or frequency of messages).

IPR2025-00200 / 722 Patent Page 38 of 92 EX1010

3) It is obviously desirable to provide full and uninterrupted service to each customer. Unfortunately, in order to protect the integrity of the network operations, it may at times prove necessary to disrupt in some manner the service to one or more customers. When this is required, the following guidelines apply:

   a) The "least bad" action is to drop messages for a customer.
   b) The "moderately bad" action is to not allow additional connections for a customer.
   c) The "bad" action is to actively shut down connections for a customer.
   d) The "very bad" action is to shut down a customer entirely.

4) In developing solution strategies, the following criteria apply:

   a) Simpler and easier to implement solutions are better than complex strategies.
   b) Solutions that can be applied at the gateway or the node level without knowledge of the state of the rest of the system are better than solutions that rely upon data that must be aggregated system-wide (the latter require monitoring solutions that are not currently available, and are difficult to deal with in real-time even when monitoring does become available).
   c) Any solution that might potentially lead to chaotic system behavior, no matter how benign it seems to be in principle, should not be implemented without verification of expected results, either through mathematical proof or through simulation.

Potential Problem Areas

There are several important areas in which system congestion may occur. They include the following:

1) Message congestion at the gateway

This occurs when the gateway is receiving messages from customers faster than it is able to distribute the messages to all nodes, and as a result the incoming or outgoing buffers become filled up. The most likely scenario that could lead to this would be if a customer had a bug in an application that caused messages to be sent through the API or the data-pump at too rapid a rate, or at too large of a size.

2) Message congestion at the node

This occurs when the node is receiving messages from gateways faster than it can distribute the messages to open client connections, and as a result the incoming or outgoing buffers become filled up. Assuming that excessive message traffic from a single customer can be caught at the gateway, then the most likely scenario that could cause this problem would be if multiple customers were all sending relatively large (though not necessarily excessive) amounts of data *at the same*

time, and the aggregate of the customer messages was sufficient to overwhelm the node.

3) **Excessive client connections**

This occurs when the number of clients that try to connect to the Bang Network exceeds the capacity across all nodes, and as a result new connections cannot be accepted by the network. Assuming that Bang Networks does not oversell the capacity of the network, then this can only occur if one or more customers exceed the contracted limits on user connections. Note that as the level of connections increases, the potential for message congestion at the node is also exacerbated, because each message that is processed must be sent to a larger number of clients.

The network safeguards to be adopted must be capable of preventing all of these potential problems.

Detecting Problems

Before it is possible to take action when one of these problems occurs, it is first necessary to detect it. Ideally, this detection should take place as far in advance as possible so as to provide the best opportunity to deal with the problem in a proactive rather than reactive manner. To help with advance detection, it is deemed appropriate to set up various data structures and tracking mechanisms within the gateway and the node. These are described as follows:

The Gateway Message Queue:

Within the gateway a message queue will be constructed. This queue will house, temporarily, messages that have been received from customers but have not yet been sent to the nodes. The general functionality of message processing within the gateway will then be as follows. As first priority, the message processing loop will try to pull any new messages from the incoming buffer and add them to the queue. If there are no new incoming messages, then the loop will take any existing messages within the queue and send them along to the nodes.

The queue itself will include the following information for each message:

- The customer that sent the message
- The time that the message was received
- The size of the message (in bytes)
- The message itself

The use of the queue provides two distinct advantages. First, it can help to prevent message overflow during short bursts of messaging activity. Essentially, if several messages arrive very close together, they can be stored in the queue temporarily until there is a lull again, at which point they can be sent along to the nodes.

Second, the size of the queue can be used to help identify potential message overflow problems over a more sustained period of time. If the size of the queue grows to a significant size (where size is measured both in terms of the number of messages and the total message bytes), then appropriate steps may be taken to prevent message overflow (e.g., dropping messages or cutting off one or more customers).

Gateway Throughput Statistics:

In addition to the message queue, with implicitly stores information about the *current* message content being handled by the gateway, it may also be useful to track statistics relevant to *recent* message activity over some defined interval(s) of time (e.g., over the past five seconds, or the past 30 seconds, or the past 60 seconds). These statistics, which should be tracked both by customer and at the aggregate level, include the following:

- Number of messages sent
- Size of each message sent
- Total volume (sum of size for all messages)

The primary advantage to tracking such statistics is that they provide information that may help to identify when an individual customer is responsible for flooding the system with messages. Based upon this information, it may be possible to *selectively* lock out the messages of one customer without impeding the messages of another, should that course of action be deemed necessary or appropriate.

The Node Message Queue:

Similar to the gateway, the node will also include a queue for housing messages (in this case, messages that have been received from gateways and not yet sent out to client connections). Again, the first priority within the message handling loop in the node should be to pull any new messages from the incoming buffer and add them to the queue. When there are no new incoming messages, then the message handling loop can remove existing messages from the queue and send them along to the client connections.

The queue itself will include the following information for each message:

- The customer that sent the message
- The time that the message was received
- The size of the message (in bytes)
- The message itself
- The number of connections to which the message will be sent

The advantages to using a message queue within the node are similar to that for the gateway. Not only will the queue allow the node to weather short bursts of intense messaging activity, but it will also serve to alert the danger of message overflow when its size grows too large. These two functions are deemed particularly important within the

node, given the fact that the gateway only needs to send a given message to (eventually) hundreds of nodes, whereas the node may often need to send a message to thousands of connections.

Node Throughput Statistics:

Just as with the gateway, it is also useful to track a variety of throughput statistics over recent intervals of time at the node level. Again, these messages should be tracked both by customer and at the aggregate level, and should include the following:

- Number of messages sent
- Size of each message sent
- Number of connections to which each message is sent
- Total volume (sum of size multiplied by connections)

These statistics provide information that may help to identify when one or more individual customers is responsible for flooding the system with messages. Based upon this information, it may be possible to *selectively* drop the messages of one customer without impeding the messages of another.

Node Connection Statistics:

A final set of statistics that is relevant to detecting potential problems at the node level pertains to the number of open connections for different customers. This information can potentially be used in several different manners. First, if the system is approaching a certain level of congestion, it may be possible to selectively refuse connections for one customer (specifically, a customer that appears to be over its connection limit) while continuing to allow those for another. Alternatively, if the system is totally impacted, this information can be used to help determine whether to drop connections for a specific customer only, or to drop connections for multiple (or all) customers indiscriminately.

Aggregate System-Wide Statistics:

In theory, it would be advantageous to track a variety of statistics, both by customer and across all customers, on a system-wide basis. These statistics would include:

- Number of messages sent over recent interval
- Size of messages sent over recent interval
- Number of connection to which messages were delivered over recent interval
- Total number of active (open) client connections

The advantage to such statistics, if they can be gathered and aggregated across the entire system in a timely fashion, is that they provide a true and accurate picture of the overall level of system utilization by each customer. Thus, if it is deemed appropriate to attempt to distinguish between compliant and non-compliant customers during periods of high

system congestion (and to apply sanctions, such as dropping messages, selectively based upon that information), this represents the most reliable source of data upon which to make such judgments.

Unfortunately, in order to collect this data it will be first necessary to develop a rather sophisticated monitoring solution, which will take some time. Furthermore, even when monitoring is in place, it still may prove difficult to gather and aggregate all of the information in real-time. Given this, it is certainly not feasible in the short run to adopt and network safeguard procedures that would rely upon this data in aggregated form, and may not even prove feasible in the long run.

Handling Problems

At this juncture, no specific solutions for handling the problems described earlier have been adopted, although a number of alternatives have been suggested. The following discussion covers each of these alternatives, and attempts to outline both the advantages and disadvantages to each.

Methods for handling excessive congestion at the gateway:

a) When queue fills, drop messages indiscriminately for all customers

Advantages:
- Easy to implement
- Unlikely to cause chaotic behavior across system

Disadvantages:
- Punishes compliant customers as well as non-compliant customers

b) When queue fills, try to determine who is at fault based on recent message throughput at the gateway and then drop messages just for offending customers

Advantages:
- Attempts to spare non-offending customers
- Unlikely to cause chaotic behavior across system

Disadvantages:
- It is tricky to determine the right time-slice of prior history to examine in order to make sure that we are really catching the offending customer
- It is possible that the jam up could be caused clogged nodes, and that the nodes are clogged because of other customers at other gateways

c) Regardless of whether the queue is filled or not, track the total volume of each customer through the gateway through a fixed time period. Make sure that this volume is not exceeded, and if it is start to drop messages.

Advantages:

- Easy to calculate
- Unlikely to cause chaotic behavior across system
- Punishes offending customers without impacting non-offenders
- Should help to prevent congestion at node as well, by limiting the amount of data that each gateway could pump

Disadvantages:

- If one customer pumps all messages through a single gateway, whereas another pumps messages through multiple gateways, then the customer using multiple gateways will be allowed to send more data to the network.
- If a strict limit is imposed at each gateway, then at the system-wide level we may end up accepting a much lower level of message throughput than would otherwise be possible.

d) Track total input rates for a customer across all gateways. If the total rate exceeds their contracted limits, shut the customer down (or alternatively, drop a subset of their messages).

Advantages:

- Most "equitable" solution
- Allows largest possible throughput for non-offenders

Disadvantages:

- Requires full monitoring system, and difficult to achieve in real-time

Methods for handling excessive congestion at the node:

a) When the message queue fills to a certain point, drop messages indiscriminately

Advantages:

- Easy to implement
- Unlikely to cause chaotic behavior across system

Disadvantages:

- Punishes compliant customers as well as non-compliant customers

b) When the message queue fills to a certain point, look at recent message throughput at the node and try to determine the offending customer. Drop messages for that customer specifically.

Advantages:

- Attempts to spare non-offending customers

Disadvantages:

- May be hard to determine exactly who the offending customer is. May depend on the time-slice. May also depend on the relative number of connections per customer at the node as compared with other nodes.

c) **Examine the total throughput across all nodes to try to determine the offending customer. Drop messages for that customer specifically.**

Advantages:

- Attempts to spare non-offending customers
- Uses the best possible data

Disadvantages:

- Determination of the exact "offender" may still be sensitive to the time-slice that is used.
- Relies on real-time aggregated data, which is not likely to be available in the near term.

d) **Drop customer connections (either discriminately or indiscriminately) as a method of reducing message congestion (remaining messages sent to fewer connections)**

Advantages:

- In theory reduces message throughput time

Disadvantages:

- Difficult to determine which connections to drop
- Any dropped connections will try to reconnect to the system, which could lead to unexpected or chaotic behavior at the system level (e.g., if many nodes start dumping connections at the same time)
- Difficult to manage "state" data that would determine when to drop connections and when to allow them to resume
- Bottom line – this is probably no a very good solution

Methods for handling excessive connections at the node

a) **When the total number of connections (across all users) reaches an upper limit, allow no additional connections**

Advantages:

- Easy to implement
- No chaotic behavior across system

Disadvantages:

- Punishes compliant customers, does not punish non-compliant customers

b) **When the total number of connections for any given customer reaches a fixed limit above which no customers are allowed to cross (e.g., no single customer can have more than 50% of the connections on a given node), then allow no further connections for that customer on the node, but keep accepting connections for other customers.**

Advantages:

- Easy to implement
- Prevents a single customer from totally hogging the node's connection limit

Disadvantages:

- Slight possibility of unexpected behavior at aggregate system level
- Does not ensure that connections shared equitably even at the node level (although it does prevent a single customer from hogging all of the connections)

c) **Determine the number of connections that each customer should receive on each machine according to their contracted levels. For example, if ESPN has purchased 100,000 connections and there are 50 nodes total, then they "own" 2000 connections per machine. Next, augment this by a small margin for error (e.g. 10%, bringing the number up to 2200). Now, across all customers, determine the total number of "reserved" connections. Subtract this sum from the total number of connections that can be supported by a node. Call the difference "excess capacity". Now, whenever assigning connections to a customer, if the customer is currently under its limit, the connection should be always made. If, in contrast, the customer is over the limit, then the connection can still be made if the excess capacity has not already been used up by the current customer and other customers. If the excess capacity has been used up, however, then the connection must be refused, for any remaining open connections need to be saved for other customers.**

Advantages:

- Seems like a reasonably fair approach

Disadvantages:

- Must update at least monthly the percentages of contracted capacity for each customer
- Difficult to predict what affects this may have on aggregate system behavior (any instabilities?)
- Still does not guarantee total equity between customers (but does come closer than prior alternatives)

d) **Similar to (c), but allow a customer to use connections that are theoretically allotted to others. Then, if the other customers need those connections back, bump the connections to the first customer (who is far over their limit) and assign the connections to the customer to which the connections were originally allotted.**

Advantages:

- We get to sell more connections

Disadvantages:

- Potentially VERY dangerous in terms of aggregate system behavior if we start bumping large numbers of connections at the same time, which must all try to reconnect at the same time.
- Requires a fair amount of difficult book-keeping

e) **Similar to (c), but based on data that is aggregated across all nodes**

Advantages:

- This is probably the most equitable solution

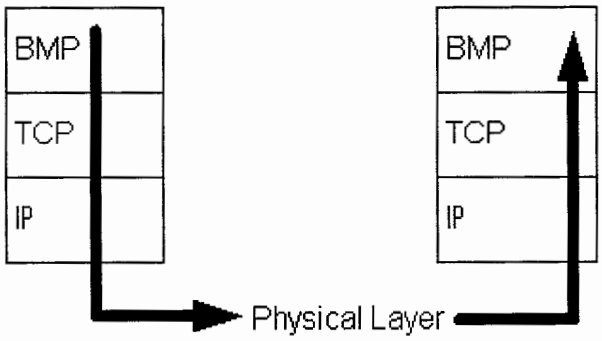
Disadvantages:

- Difficult to aggregate and use this data in real time

Bang Message Protocol

Overview

The Bang Message Protocol (BMP) is a system of messages designed to move information around the Bang Network. It is a "high-level protocol" in the sense that it sits above host-to-host transport protocols such as TCP and UDP. (In the current implementation of BMP, only TCP is used.)



BMP Header

A BMP message consists of header and body parts. The body contains the message itself and the header contains information the receiving system needs to interpret the message. BMP messages have the following form.

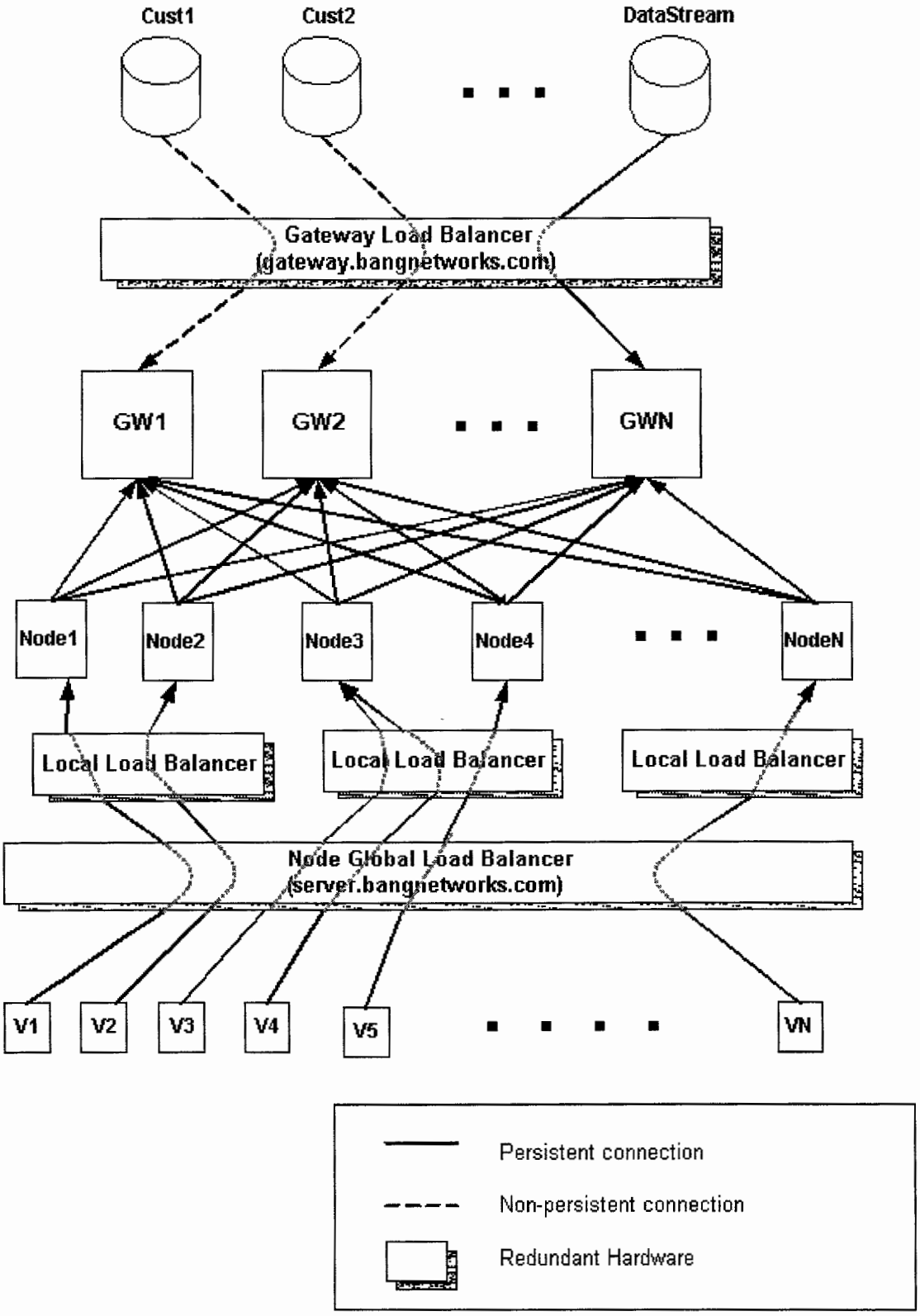
MESG_TYPE_TOKEN	END_TYPE_TOKEN	BODY_LENGTH	END_TOKEN	BODY
VARIABLE	1 BYTE = SPACE	VARIABLE	2 BYTES = CRLF	BODY_LENGTH

To distinguish between the different types of BMP messages, the header contains a MESG_TYPE_TOKEN. Each message type has an associated integer (defined in the `BMPMessageConstants` interface) and this is placed in the first field of the header. Since this integer has variable length, the next field is an END_TYPE_TOKEN (defined in `BMPMessageConstants` to be `SPACE`). Next comes the BODY_LENGTH, which, naturally, contains the length of the message body (in bytes). Since this field is also variable length, it is followed by the END_TOKEN (= `CRLF`).

A BMP message with zero-length body is allowed. In this case the message header itself is the message.

If a BMP message is authenticated, it will also contain a 20-byte appendix. For details, see "Authentication" below

Architecture



Identifiers

The Bang Network uses a system of identifiers to locate network elements, track data, and enable authentication. For the most part, these identifiers are quite straightforward: each node has a Node ID, each gateway has a Gateway ID, and each customer has a Customer ID. In addition, each live-enabled object on a customer's web site has an Object ID (called a Bang ID in the product documentation). In fact, an html object is marked as "live" by including the phrase "id=Bang$<object id>" in its html tag.

The Batch ID is a more subtle type of identifier. Technically, a Batch ID is just an Object ID used in a different context. When a customer wants to send messages in batch mode (see `BatchBroadcastCustomerData` below), it hides the Object IDs for the individual messages in such a way that they are invisible to the gateways and nodes. In order for a node to determine which customers need to receive the batch message, the customer includes a Batch ID that is really just the Object ID for some object on the page. Often the html tag <body> is marked as live for just this purpose, though a Batch ID can be the Object ID for any object that all viewers needing the updates will have registered. (For the concept of registering objects, see the `ViewerConnectionRequest` below.)

Authentication

Every communication between a gateway and a node is authenticated. Also, every message sent from a customer to a gateway is authenticated. To facilitate this authentication, each customer, gateway, and node has a secret key. When a Sender wants to send an authenticated message, it first inputs its secret key and the BMP message to the Secure Hashing Algorithm (SHA). The output is a 20-byte digest that the customer attaches to the message as an appendix.

In order to authenticate the message, the Receiver must know the Sender's secret key. For this purpose, every gateway maintains a list of the secret key associated to each Customer ID and Node ID. Nodes maintain lists that associate the appropriate secret key to each Gateway ID.* Upon receiving a message, the Receiver reads the BMP message as well as the digest. It then looks up the Sender's secret key and computes its own copy of the digest by applying the SHA to the Sender's secret key and the received message. If this digest is identical to the received digest, the message is authenticated.

At the moment, messages between nodes and viewers are not authenticated. One option for the future would be to establish SSL connections with viewers.

A weakness of the present authentication scheme is the requirement that gateways and nodes maintain lists that associate secret keys to IDs. For example, at the moment there is no mechanism for adding new entries to this list.

---

* Since customers never receive authenticated messages, customers do not need to know any secret keys (except their own). In fact, since `GWCustomerConnectionRequest` is not currently used, customers receive no application layer messages at all.

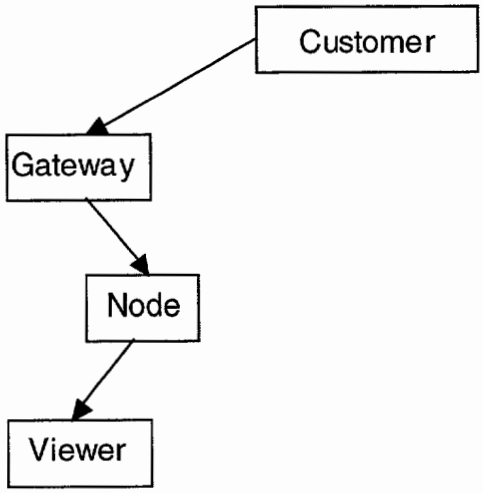
For more detail on authentication, see the "Overview of Authentication" document. However, the information there is slightly out of date. In particular, note that timestamps are NOT currently used in the authentication process.

Message Types

We will divide message types into three categories according to their function: broadcast, initialize, and administrative/miscellaneous. These are rough and informal divisions.

For complete documentation on the fields for these message types as well as the serialization process, see the Appendix.

Broadcast Messages



The basic function of the Bang Network is to transmit messages from customers to viewers. These messages can be sent in either regular or "batch" mode in which several messages are wrapped together and sent as a single message.

A message is sent from a customer to a gateway using the `BroadcastCustomerData` and `BatchBroadcastCustomerData` BMP message types. The gateway then forwards the message to each node using the same BMP message type. Finally, the nodes send the message to the viewers that need it using the `BroadcastViewerData` or `BatchBroadcastViewerData` BMP message types.

In order for a node to determine which viewers need a particular message, it must know which customer (e.g. ESPN) the message came from, and which particular item on that customer's web site (such as the score in the Raider's game) the message relates to. The

node will send the message only to viewers that are displaying that particular item. For this reason, the `BroadcastCustomerData` BMP message type includes **customerID** and **objectID** fields, in addition to the **JSStatements** field, which contains the message itself (such as some JavaScript that updates the Raider score).

On the other hand, each connection between a viewer and a node can refer to only one customer ID. Therefore, the node does not need to forward this information on to the viewer, and the `BroadcastViewerData` BMP message type includes only **objectID** and **JSStatements** fields, and not a **customerID** field.

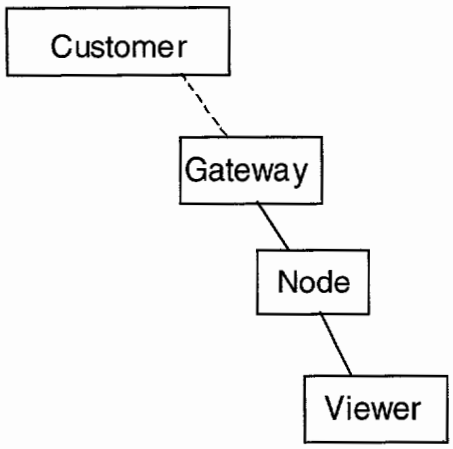
Instead of sending a number of individual `BroadcastCustomerData` messages, a customer can choose to package several messages together using the `BatchBroadcastCustomerData` BMP message type. The important fields in this message type are **customerID, batchID,** and **messages**. The **messages** field contains a vector of messages where each message is simply a `BroadcastCustomerData` object with the **customerID** field removed. The **batchID** is the object ID for any object on the page to receive the messages. For details see "Identifiers" above.

Since at the moment a separate connection between a customer and a gateway needs to be established for each message, the main advantage of using batch mode is that several messages can be sent using a single connection. In addition, the customer ID does not get sent multiple times.

When a `BatchBroadcastCustomerData` message arrives at a gateway, the gateway passes it on to the nodes. The nodes strip off the customer ID – thereby converting it to a `BatchBroadcastViewerData` – and send it to the viewers that need it. This process is exactly parallel to the procedure with `BroadcastCustomerData`.

Initialize Messages

For the most part, connections between viewers and nodes and between nodes and gateways are persistent, but connections between gateways and customers are not persistent.



However, in the future we may want to allow customers to establish persistent connections with gateways. A customer would initiate this by sending a `GWCustomerConnectionRequest`, which includes a **customerID** field. The gateway would respond with a `GeneralAcknowledgement`. This is an example of a BMP message type that includes no fields.

A basic design principle of the Bang Message Protocol is that gateways should not initiate connections with other network elements. A connection becomes established when a gateway *receives a request* for a connection and *acknowledges* it. This is the procedure for customer connections, as we just discussed. Connections between gateways and nodes work in the same way. When a node wants to connect to a gateway, it sends a `GWConnectionRequest` with its node ID in the **ID** field. Since all communication between gateways and nodes needs to be authenticated (see above), the gateway responds with an `AuthenticatedGeneralAcknowledgement`. (NB the name of this message has recently changed; this is the new name.)

To understand the Bang Network, it is important to be clear about what is meant by a "viewer." When an end user loads a web page with live-enabled content, an applet (sometimes called the "Bang Client") loads and tries to connect to the Bang Network.* This applet then becomes the network element we call a viewer. If an end user opens more than one browser window, more than one instance of the applet will load. In that case, a single end user would correspond to multiple viewers, each of which is an independent network actor. A similar situation could arise with web pages that contain

* The process is actually a little more complicated than that. When an end user loads a web page with live enabled content, it is told to download an applet from server.bangnetworks.com. The load balancer at this address passes the request to our applet server, which sends the applet. For security reasons, an applet is only allowed to make a connection to the URL from which it was downloaded. But when the applet tries to make the connection back to the applet server, our load balancer intercepts the request and redirects the connection to one of the nodes. If the applet was downloaded directly from the customer, it would only be able to establish a connection to the customer and there would be no simple way to direct it to the Bang Network.

multiple frames. Each frame could act as a separate browser window and launch its own instance of the applet.**

A viewer requests a connection by sending a BMP message of type `ViewerConnectionRequest` to a node. The viewer needs to tell the node which customer's page originated the applet, and to register the live-enabled objects for which it should receive updates. Therefore, a `ViewerConnectionRequest` includes a **customerID** field and also an **objectIDs** field. The latter is an array of object IDs. For example, if an end user downloads a page from ESPN.com with two live enabled objects, say the Raiders score and the 49ers score, then it would send ESPN's customer ID and the Object IDs for these two objects. These objects are now registered to that viewer.

The node responds to the `ViewerConnectionRequest` with a `GeneralAcknowledgement`, and will now monitor the `BroadcastCustomerMessages` it receives for messages from this customer (such as ESPN). When it receives one, the node checks the Object ID (or BatchID in the case of `BatchBroadcastCustomerMessages`) to see if the viewer has registered that ID. If so, it forwards the message to the viewer.

Another type of initialization is necessary when a new node or gateway is brought on line. Nodes should know the addresses of the gateways so the first task for a node is to connect to the gateways. A node requests a connection to a gateway by sending it a `GWConnectionRequest` containing its Node ID in the **ID** field. The gateway responds with an `AuthenticatedGeneralAcknowledgement`.

The process for a new gateway is more complicated. Initially, a new gateway does not know the address of any other gateway or any node. Also remember that gateways prefer to *receive* requests for connections, not *make* requests for connections. The first task for a new gateway is to send the load balancer a BMP message of type `GWAlive`. This message includes the **host** (name) and **port** for the new gateway as well as the **newGatewayID**. Unless this is the first gateway, the load balancer passes the message to an existing (old) gateway. The old gateway welcomes the new gateway by sending an `AuthenticatedGenericMessage` that includes in its **msg** field the number of

---

** Since creating lots of viewers is an inefficient use of network resources, the Bang Network API defines techniques that web sites can use to avoid creating multiple viewers when using frames. (On the other hand, since customer ID information never reaches a viewer, in any situation where live-enabled content comes from two different customers, there must be two different viewers.) Another complicating factor is that current implementations of Internet Explorer allow only three TCP connections per thread. (Clicking on the IE icon a second time produces a new thread, but opening a new window with "<control>n" does not.) For all of these reasons [and maybe some involving firewalls?] we may expand the applet's functionality along the following lines: when the applet is downloaded from a particular customer's web site, the applet first checks for an existing viewer that is listening for live updates from that customer. If such a viewer exists, the applet tells it to add the new BangIDs to the list of items it is listening for. (See `SubscribeObjects` below.) If there is no current viewer waiting for updates from that customer, the applet creates one.

nodes. When the new gateway is ready to establish connections with these nodes, it sends a `GWAliveSendNodeConnection` to the old gateway that then passes the message on to all the nodes as a `GWAlive` message. When a node receives a `GWAlive` message it asks for a connection with the new gateway in the usual way, that is by sending a `GWConnectionRequest`.

For more detail on the gateway initialization process, see the document "Functional Specification for Gateway Fault Tolerance."

Administrative/Miscellaneous Messages

The administrative portion of the Bang Message Protocol is less well developed than the broadcast and initialization portions. Currently there are only two message types, `InfoRequest` and `GenericMessage`. `InfoRequest` is sent from an administrative computer to a gateway or a node. It is just a general request: there are no fields specifying the type of information requested. The gateway or node sends its status in the **msg** field of the `GenericMessage` BMP message type.

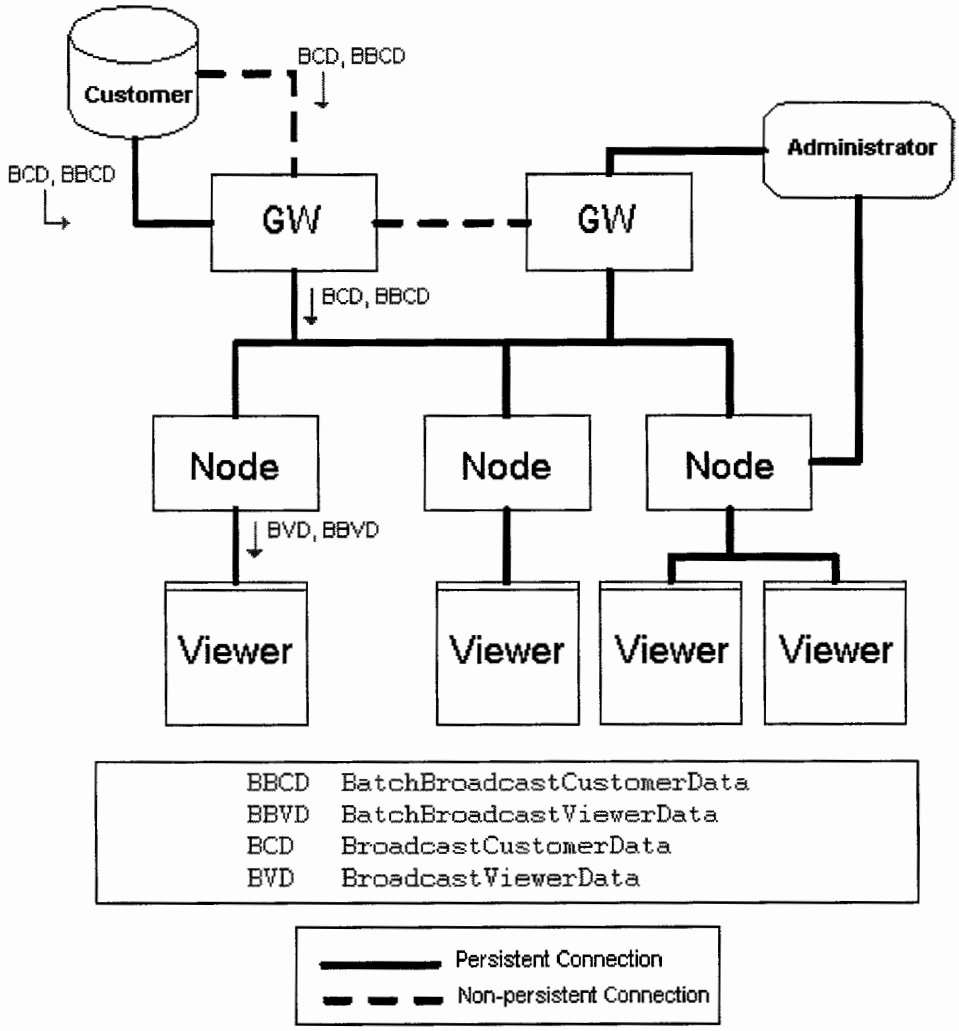
At the moment, once a viewer has sent a `ViewerConnectionRequest`, there is no way for it to change the ObjectIDs it has registered with the node. If an end user wants to receive updates for a different set of objects, the connection must be closed (i.e. the viewer must die) and a new connection established, even if the objects come from the same customer. In the future, however, we may implement a BMP message of type `SubscribeObjects` that would send a new set of **objectIDs** to the node. [One of the reasons this is tricky is that the method we currently use to establish the connection does not allow the viewer to send application-layer data once the connection is established.]

There is one miscellaneous message type that is important in some situations. Normally, when one side wants to close a connection, it directs TCP to issue an active close. However, when TCP issues an active close, the TCP virtual circuit is put into a waiting state while the requesting machine waits for a (TCP-layer) acknowledgement. Until this acknowledgement arrives, the port cannot be reused. For this reason, it is sometimes desirable for one side to ask the other side to issue the active close. The Bang Message Protocol provides for this with the BMP message type `ConnectionClose`.

`ConnectionClose` is used, for example, if a new gateway does not get as many connection requests from nodes as it expects. The new gateway sends `ConnectionClose` messages to each node it has already heard from, and then starts its initialization process over again. Another possible use for `ConnectionClose` would be for a gateway to ask a customer to terminate a persistent connection (if `GWCustomerConnectionRequest` is implemented and so this type of persistent connection becomes available).

Appendix: BMP Reference Guide

Broadcast Messages



Details:

In what follows, "FST" stands for "Field Separation Token" and "GST" stands for "Group Separation Token." They equal ascii values 28 and 29 (in decimal), respectively.

BatchBroadcastCusotmerData

Used to send a collection of broadcast messages from a customer to a gateway or from a gateway to a node.

FIELD	customerID	token	batchID	token	numMessages	token	messages
CONTENTS	customer ID	FST	batch ID	FST	numMessages	FST	

FIELD	messages					
CONTENTS	objectID	token	javascript	token	isRawJavascript	token*
	objectID	GST	Javascript	GST	IsRawJavascript	FST

*FST appears at the end of every message except the last one.

BatchBroadcastViewerData

Used to send a collection of broadcast messages from a node to a viewer.

FIELD	batchID	token	numMessages	token	messages			
					objectID	token	javascript	token*
CONTENTS	batch ID	FST	numMessages	FST	objectID	GST	javascript	FST

*FST appears at the end of every message except the last one.

BroadcastCusotmerData

Used to send a broadcast message from a customer to a gateway or from a gateway to a node.

FIELD	customerID	token	objectID	token	isRawJavascript	token	JSStatements
CONTENTS	Customer ID	FST	object ID	FST	A flag which tells whether the incoming Javascript should be executed in the raw or as a method of an object	FST	Javascript statements for this message

---

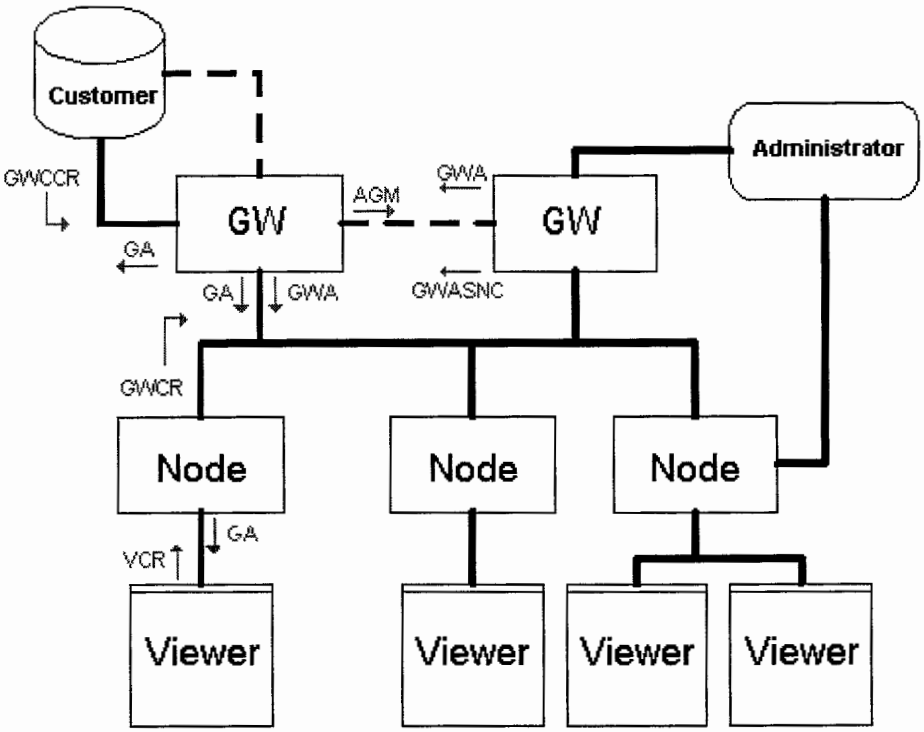
BroadcastViewerData

Used to send a broadcast message from a node to a viewer.

FIELD	objectID	token	isRawJavascript	token	JSStatements
CONTENTS	object ID	FST	A flag which tells whether the incoming Javascript should be executed in the raw or as a method of an object	FST	Javascript statements for this message

---

Initialization Messages



AGM	AuthenticatedGenericMessage
GA	GeneralAcknowledgement
GWA	GWAlive
GWASNC	GWAliveStartNodeConnections
GWCR	GWConnectionRequest
GWCCR	GWCustomerConnectionRequest
VCR	ViewerConnectionRequest

Persistent Connection
Non-persistent Connection

Details:

In what follows, "FST" stands for "Field Separation Token" and "GST" stands for "Group Separation Token." They equal ascii values 28 and 29 (in decimal), respectively.

`AuthenticatedGenericMessage`

Used when an existing gateway responds to a `GWAlive` message announcing a new gateway coming on line. It includes the number of active nodes in the **msg** field. Could be used any time information needs to be sent that does not fit within another BMP message type and needs to be authenticated.

FIELD	ID	token	msg
CONTENTS	ID of the originator of the message	FST	message string

---

`GeneralAcknowledgement`

This message type has no fields. It is used by a gateway to respond to a request for a connection from a customer (`GWCustomerConnectionRequest`) or a node (`GWConnectionRequest`), or by a node to acknowledge a request for a connection from a viewer (`ViewerConnectionRequest`).

`GWAlive`

Used by a new gateway to inform an existing gateway that it is alive, or by the existing gateway to inform the nodes that a new gateway is alive.

FIELD	host	token	port	token	newGatewayID
CONTENTS	Name of the new gateway	FST	Port the new gateway is listening on	FST	ID of the new gateway

---

`GWAliveStartNodeConnections`

Once a new gateway has announced its existence (`GWAlive`) and received the number of nodes in return (`AuthenticatedGenericMessage`), it prepares to establish connections with the nodes and then sends a `GWAliveStartNodeConnections` message to the existing gateway indicating that it is ready for the connections to begin. The fields for this message are identical to `GWAlive`.

FIELD	host	token	port	token	newGatewayID
CONTENTS	Name of the new gateway	FST	Port the new gateway is listening on	FST	ID of the new gateway

---

`GWConnectionRequest`

Used by a node to request a connection with a gateway.

FIELD	ID
CONTENTS	ID of the requesting node

`GWCustomerConnectionRequest`

Used by a customer to request a connection with a gateway.

FIELD	customerID
CONTENTS	ID of the requesting customer

`ViewerConnectionRequest`

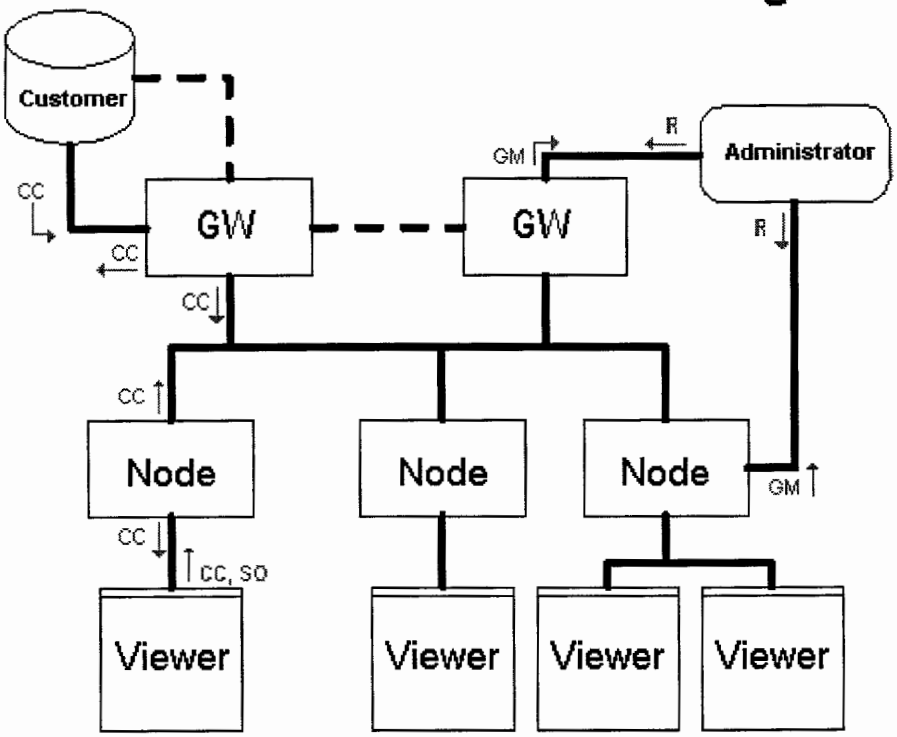
Used by a viewer to request a connection with a node and to register object IDs (also called Bang IDs).

FIELD	customerID	token	numIDs	token	objectIDs	
					objectID	token*
CONTENTS	ID of the customer who owns the Bang IDs that the viewer is registering	FST	numIDs	FST	objectID	GST

*GST appears at the end of every message except the last one.

IPR2025-00200 / 722 Patent Page 61 of 92 EX1010

BMP Miscellaneous Messages



CC	ConnectionClose
GM	Generic Message
IR	InfoRequest
SO	SubscribeObjects

Persistent Connection

Non-persistent Connection

Details:

ConnectionClose

This message type has no fields. It can be used by any element to request that the other side of a persistent connection issue a TCP active close. In particular, when a new gateway does not receive as many `GWConnectionRequests` as it expects, it sends `ConnectionClose` messages to the nodes with which it has successfully connected before restarting its initialization sequence.

GenericMessage

Can be used to send any string message. In particular, used to respond to an `InfoRequest`. (Note that `InfoResponse` is no longer used for this purpose.)

FIELD	msg
CONTENTS	generic string message

InfoRequest

This message type has no fields. It is used when an administrative machine wants to ask a network element for its status.

SubscribeObjects

Used by a viewer to update the list of object IDs it is looking at. Currently this BMP message type is not used since the present viewer technology does not allow upstream messages. At the moment, if the viewer needs to change the list of objects it has registered, it must break the connection and re-establish it using another `ViewerConnectionRequest`.

FIELD	objectIDs
CONTENTS	Array of object IDs that the viewer is looking at

Proposal for Financial Data Stream

Goal: deliver 10K quotes per second to 50K viewers, each viewer gets 25 quotes

Problems:

- Strains network elements, e.g. node memory
- Bandwidth costs very high

Summary of proposal: send all the quotes in one large message through a gateway to the nodes; the nodes disassemble the message and deliver a single message to each viewer containing the 25 quotes that the viewer needs.

This approach uses minimal bandwidth at the cost of extra strain on the nodes. However, by carefully choosing the data structures involved, the impact on the nodes can be reduced.

Upshot: could send 40 bytes of data for each stock (excluding the name) for 760Mbps – including all traffic from customer to gateway, gateway to nodes and nodes to viewers, and all headers and acknowledgements.

Data Structures:

Currently, Bang uses two layers of identifiers to route data: Customer IDs and Object IDs. It would be natural to assign a unique Object ID to each stock quote. However, it also makes sense to group these objects together in some way. There are several equivalent ways to do this but I propose to introduce another layer of identifier *below* Object ID. Then the collection of all stock quotes would fall under a single Object ID and the individual quotes would correspond to subObject IDs. (It would also be possible to break the collection of all quotes into several baskets, each with its own Object ID; these baskets would be updated separately.)

Two important things about subObjectIDs:

- They must be numbered sequentially. For example, under the object "Stocks," there might be subobjects 0000, 0001, 0002, …, 9999.
- Customer web pages can reference subobjects just as they do objects. We could, for instance, make the convention that a Bang ID of the form "Bang$OBJ%NUM" refers to subobject NUM of object OBJ.

When a customer wants to update an object that is made up of subobjects, it sends a message that contains its Customer ID, the Object ID (such as "Stocks") and a vector, called a Customer Update Vector, that has one entry for each subobject containing the new data for that subobject. Such a message would be a new BMP message type, perhaps "subObjectCustomerData." For example, suppose that AOL is subobject 1234 in the "Stocks" object. Then, when updating the "Stocks" object, entry 1234 of the Customer Update Vector would contain the AOL stock quote – perhaps two numbers representing bid and ask prices.

In order to implement subobjects, there must be minor changes to the applet and the node. When an applet sees a page containing Bang IDs of the form "Bang$OBJ%NUM," it does not register them individually. Instead, it registers the Object ID OBJ, along with an array, called a Viewer Registration Array, containing the NUMs. The NUMs appear in the Viewer Registration Array in the order they appear on the page. For example, if my screen is displaying quotes for "Stocks%0567," "Stocks%9981," "Stock%4567," etc., then my applet would register the object "Stocks" along with the Viewer Resistration Array [0567,9981,4567,...].

The node must have a mechanism for storing these Viewer Registration Arrays. Currently, nodes maintain a hash table that maps a pair (Customer ID, Object ID) to a linked list of viewers. Now, the linked list will contain pairs (Viewer, Viewer Registration Array).

When a subObjectCustomerData message arrives at a node, the node hashes (Customer ID, Object ID) and then starts marching down the linked list. For each element, it creates a new vector called a Viewer Update Vector. This vector is created by taking those entries from the Customer Update Vector that correspond the entries of the Viewer Registration Array. For example, suppose the Customer Update Vector is

[update to subobject 1, update to subobject 2, update to suboject 3, ...],

and the Viewer Registration Array is

[1234,0991,1010],

then the Viewer Update Vector would be

[update to subobject 1234, update to subobject 0991, update to subobject 1010].

Finally, the node would send a message to the viewer (of type subObjectViewerData, say) that contains the Object ID and the Viewer Update Vector.

When an applet receives a message of type subObjectViewerData, it applies the first entry of the Viewer Update Vector to its first subobject of the Object ID, the second entry to the second instance of the Object ID etc.

Analysis:

Impact on Nodes – Memory: The memory cost of storing Viewer Registration Arrays is smaller than the present alternative, namely registering each stock as a separate object. (Instead of storing 25 (present model) linked list elements, we store one (present model) linked list element and an array of 25 numbers. Plus, since we remember fewer objects, we reduce the number of Object IDs stored in memory.) On the other hand, under the new model, when an object doesn't have any subobjects, we need to store an empty array – probably a null pointer, actually – which adds to overhead. (E.g. if 10K viewers look at 100 objects each, one 4 byte null pointer would at 4 megabytes.) [Another issue: sending fewer messages (one instead of 25) reduces memory costs associated with temporary variables, but I don't know how to quantify it.]

Impact on Nodes – Speed: The process of forming the Viewer Update Vectors should be very fast. I anticipate that this system would be faster than handling each quote as a separate object, but I have not quantified it.

Impact on Bandwidth: See spread sheet.

Disadvantages:

- Every subObjectCustomerData message must contain information about every subobject, even if that subobject hasn't changed.
- Not especially flexible.
- Requires some intelligence at the customer.
- Requires substantial development.

Alternatives:

- Each stock is its own object and each message contains only one update (current system).
  - Advantages:
    - Very flexible
    - Does not require additional development.
  - Disadvantages:
    - Very high overhead bandwidth use
    - A lot of work for the nodes
- Each stock is its own object and messages are batched.
  - Advantage:
    - Lower overhead bandwidth use
    - Does not require additional development.
  - Disadvantages:
    - Stock updates sent to viewers that don't need them (could add substantially to bandwidth use)
    - Requires intelligence at the customer.
- Use a system similar to the one described, except instead of using fixed length Update Vectors, use a variable length mechanism.
  - Advantage:
    - Flexible – don't have to update every subobject with every message
  - Disadvantages:
    - Must use bandwidth identifying which stocks are being updated (e.g. by including subObject IDs).
    - Slower at the node because need work with a variable-length data structure.
    - Requires some intelligence at the customer.
    - Requires substantial development.

Other Applications:

Although the new mechanism of this proposal is primarily designed for stock quotes – and sacrafices flexibility for better performance in this area – other applications are possible. In general, applications that are hightly personalized (e.g. every individual wants at a different set of quotes) and consist of many pieces of data that are updated at the same time, are candidates for using subobjects. It is hard to say how many such applications there are, but here is one example in the area of ad serving:

> Suppose that I divide my ads into 100 categories. At each moment, one ad from each category is "current." Using some personalization mechanism, I display a different set of 5 ads to each visiter. Every 20 seconds I want to change which ads are current. I could accompish this by creating one object, "Ads," containing 100 subobjects, one for each category.

22726/05892/SF/5044451.1

OBJECT-BASED ROUTING NETWORK FOR UPDATING WEB CONTENT

BACKGROUND

FIELD OF THE INVENTION

This invention pertains in general to transferring information through digital networks and in particular to transferring information for remotely updating displays of destination devices through the digital networks.

BACKGROUND ART

The Internet is a digital network. An individual computer on the Internet is typically identified by an internet protocol (IP) address. A computer on the Internet sends a packet of information to another computer by routing the packet to the destination computer's IP address. The destination computer typically interprets the packet according to one of several possible protocols. Usually, the computer determines the protocol by determining the logical "port" to which the packet was sent.

The World Wide Web (the "Web") is a collection of technology and content available on the Internet that allows the content to be routed to particular destination computers. The Web includes a large number of files residing on many different computers (called web servers). These files are organized into units called web pages, which contain one or more files specifying instructions for presenting the web page and content, such as text, images, applets, video, and/or audio.

Web pages use a variety of definitional and programming languages to control how information is presented. The most fundamental of these is the Hypertext Markup

Language (HTML). HTML uses a system of "tags" to specify how content should be displayed. Recent advances in HTML introduce "style sheets" which help separate content information from display information. HTML has been modified and extended to provide new capabilities. For example, Extensible Markup Language (XML) adds semantic content to web pages. In addition, Dynamic HTML (DHTML) adds some dynamic content to web pages.

A web page may also include one or more programs for controlling how the web page is displayed. For example, Java applets and JavaScript scripts may be used to control the display of a web page. In addition, DHTML uses scripts to control the dynamic content. Thus, a web page designer can use applets and scripts to produce animation effects or modify the display based on user interaction. For example, the designer can write a script that changes the color of a piece of text when a user clicks on a button.

Devices that display/execute web pages are often called "client devices." Client devices include personal computers, web-enabled televisions, cell phones and other handheld devices, and special-purpose web-browsing appliances. Client devices typically employ a program called a "web browser" for interpreting the HTML or other display instructions in the web page and displaying the content accordingly. Most browsers include special functionality, such as a Java Virtual Machine, for executing Java applets and/or other applets or scripts embedded in the web pages.

A client device specifies a web page or other document on the web using a Uniform Resource Locator (URL). A URL has the form "service://server/path/file." Here "service" refers to the protocol to be used such as the file transfer protocol (FTP) or

the hypertext transport protocol (HTTP). "Server" is the IP address of the server that contains the page, and "path/file" specifies the particular web page on that server.

The Web suffers from a substantial limitation with respect to dynamically updating content in a web page at a client device. The Web's only mode of operation is for a client device to request a page from a server and for the server to send the requested page to the client device. Once the server delivers the page to the client, it typically terminates its connection to the client, and does not retain any information about the client, or the page that was sent; for this reason, servers are "stateless." As a result, client devices control the flow of information around the Web. While this control is appropriate in some situations, it does not permit efficient updating of data at the client devices. For example, if a web page contains information that may change, such as the score of a baseball game, the server has no way to inform the client devices that are viewing that page of the change. Instead, the client must ask the server for the updated information. However, the client device does not know when the information on the web page has changed, and thus does not know to ask for the updated page.

There are some simple web programming techniques that attempt to update content on client device-side web pages. One approach that web designers currently use is to rely on the client devices to periodically re-request web pages. This updating can be performed as the result of user action (such as pressing the "refresh" button) or can be automated to occur on a particular schedule (such as by using the HTML Meta Refresh tag to cause the client to request the page every 'X' seconds). Although this technique provides client devices with more up-to-date information, it is very wasteful of resources. In particular, the web server must resend the page even if nothing has changed, and, even

when something has changed, it must resend the entire web page rather than just the updated information, which may be only a very small part of the page.

The performance of automatic refreshing can be improved somewhat by putting information that may change in a separate frame than the information that is less likely to change, and only refreshing the separate frame. A few web designers even write custom Java applets to limit refreshing to individual components on a page, such as the score of a soccer game. A willingness to go to such effort illustrates the serious drain of resources caused by frequent refreshing.

Nevertheless, even custom Java applets are not a meaningful attack on this problem. Custom applets approach require a large separate development effort for each item on each page that might need to be updated. More importantly, such custom applets still update content based upon client-driven requests. This solution is not scalable to provide updated information for large numbers of client devices and for large numbers of web pages.

Therefore, there is a need in the art for an efficient way to provide dynamic content to a web-page at a client device.

BRIEF DESCRIPTION OF THE DRAWINGS

FIGURE 1 is a high-level block diagram illustrating messages passed according to an embodiment of the present invention;

FIGURE 2 is a high-level block diagram illustrating a network architecture for providing the functionality described above with respect to FIG. 1 according to an embodiment of the present invention; and

FIGURE 3 is a high-level block diagram of a computer system for use as a server, client, and/or for one or more of the various components illustrated within the network according to one embodiment of the present invention

The figures depict an embodiment of the present invention for purposes of illustration only. One skilled in the art will readily recognize from the following description that alternative embodiments of the structures and methods illustrated herein may be employed without departing from the principles of the invention described herein.

DETAILED DESCRIPTION

FIG. 1 is a high-level block diagram illustrating the communication flow according to an embodiment of the present invention. The present invention solves the problem identified above by providing a mechanism for a server to update content displayed at a client device (often referred to as simply a "client"). The present invention is highly scalable because it does not require the server to maintain state information for, or connections with, each of the clients that may be viewing one of its pages.

The server 110 is representative of a web server holding web pages for an information provider. In practice, the present invention may operate with many different servers holding web pages for many different information providers; thus a single server 110 is shown for ease of description. Briefly, an information provider is any entity desiring to send dynamically-updated web content to clients. Web pages on the web server are preferably developed using normal technologies, including HTML, Java, JavaScript, and/or any other well-known technologies. In addition, the web pages can

have any content typically found on web pages, such as text, images, streaming and/or non-streaming video, audio, etc.

A web page developer or other entity developing web pages for the information provider can designate certain objects on a web page 114 as "live." As used herein, an "object" is any datum or data in the web page that can be treated as a single item. Examples of objects include text characters and strings, images, frames, tables, audio, video, applets, scripts, variables, or other information used by applets and/or scripts, URLs embedded in the web page, etc. In addition, an entire web page or collection of web pages can be a single object. Thus, an object can be defined at any desired or useful level of granularity. In one embodiment, the developer designates an object as live by including a unique identifier for the object, called an "object identifier" or "object ID," in the web page 114.

More specifically, in one embodiment an information provider "live-enables" a web page by using the "ID" attribute supported by many HTML tags to assign an object ID to an object. The ID attribute is defined using the expression:

ID = "elementIdentifier",

where "elementIdentifier" is a string. In one embodiment, an information provider indicates a live object by making "elementIdentifier" a string beginning with a predefined token. For example, all elementIdentifiers can begin with the string "Bang$", or any other agreed upon token. For example, the following HTML code live-enables an image with the object ID "Bang$MyID":

```
<IMG SRC="someImage.jpg" ID="Bang$MyID" HEIGHT=90 WIDTH=120>
```

The "MyId" portion can be any identifier desired by the information provider. It is anticipated that each information provider will use its own unique identification codes for each of its objects. However, it is not a requirement that each MyID portion be globally unique across all information providers. Since in principle different information providers could use the same object ID, each information provider is preferably assigned a unique information provider ID and the combination of the information provider ID and the object ID is used to identify a particular object. The information provider preferably includes its information provider ID in web pages provided to the clients. Information provider IDs are also preferably used for billing and authentication purposes.

In alternative embodiments of the present invention, the system of IDs is organized differently. For example, instead of using IDs on the level of information providers and individual objects, one embodiment assigns IDs at the page level. In such an embodiment, individual objects within the page might not be marked at all. Alternatively, one embodiment organizes IDs into a deeper hierarchy, such as information provider X, data feed Y, and object Z. In yet another alternative, objects are marked "live," but are not assigned individual identifiers.

The web page may also contain initial values for the live objects. For example, a web page 114 showing scores for football games may include a list of games and the current scores as of the time when the server 110 received the client 112 request. The games and scores may additionally be designated as live objects and each have unique object identifiers. Alternatively, the live objects can be distributed without initial values.

The web page 114 also preferably includes an instruction causing the client 112 to connect to a network 116 that manages the live content. In one embodiment, the

instruction comprises the following HTML code and is placed in the web page below all of the object ID tags (assume the information provider ID is "MyinformationproviderID" and the network IP address is "BangAddress"):

<SCRIPT>SCR="http://BangAddress/MyInformationproviderID/
bangTag.bng></SCRIPT>

This HTML code causes the client 112 to download the applet "bangTag.bng" from the network 116. In an alternative embodiment, this applet is directly provided to the client 112 by the information provider.

The client 112 preferably requests and receives the web page 114 containing one or more object identifiers from the server 110. Embodiments of the present invention may have thousands or millions of clients, of which the illustrated client 112 is merely representative. In one embodiment of the present invention, the client 112 is a computer system including a display device. The computer system preferably executes a browser, such as Internet Explorer available from Microsoft Corp. The browser preferably includes a Java Virtual Machine (i.e., the browser can execute Java applets), Dynamic HTML capabilities, such as support for JavaScript (or another scripting language, such as VBScript) and the Document Object Model, and supports a technique for enabling communication between Java and the scripting language. The standard technology that allows Java and JavaScript to communicate is called LiveConnect. In some embodiments, the functionality of the browser is extended through software "plug-ins" such as Macromedia Flash, Real Networks RealAudio/Video, and/or Apple QuickTime. If so, then a developer developing web pages for such browsers can preferably designate the associated content types (e.g., Flash animations) as live content. The client may also

include a handheld personal digital assistant (PDA), such as a Handspring Visor, Compaq iPAQ, Hewlett Packard Jornado, or the like, which may communicate with the server via wired or wireless interfaces. Cellular telephones with wireless web browsers (e.g., Samsung's SCH 850) are also suitable client devices.

As described above, the web page 114 received by the client 112 from the server 110 preferably includes an instruction causing the client to connect to the network 116 providing the live content. Alternative embodiments may use different techniques for causing the client 112 to communicate with the network 116. For example, the functionality of the instruction can be integrated into the browser itself, either as native code or as a plug-in. Preferably, the network connection is established in a manner that is transparent to a user of the client and remains in effect until the client visits another web page. Alternatively, the functionality of the instruction may be provided by a standalone application on the client device.

In an alternative embodiment of the present invention, the client 112 does not download the initial web page from the server 110. Instead, functionality (e.g., an application) is already present at the client 112 to contact the network 116 and register an interest in certain objects. In addition, the network 116 may provide information about objects after the client 112 contacts it in order to assist the client 112 in selecting objects to register.

After making this network connection, the browser of the client downloads and executes an applet from an applet server portion (not shown in FIG. 1) of the network 116. In one embodiment, the applet has three functions: register object IDs with the network 116, handle messages received from the network, and open and close the

connection between the client device and the network. In order to register object IDs, the applet first parses the web page the client received from the server 110 and makes a list of all object IDs it finds. In an alternative embodiment, the applet may list only a subset of the object IDs, such as only the IDs of objects that are currently being displayed on the web page. Alternatively, the list of object IDs may be pre-encoded in the web page in addition to the objects themselves, for simple identification by the applet.

Next, the applet opens a connection to the network 116. In some cases, the client 112 may be located behind a firewall. This firewall puts a restriction on the types of connection requests the client can make – it may, for example, block all non-HTTP traffic. For this reason, the applet preferably wraps its connection request in an HTTP header. More precisely, the applet needs to communicate a vector (e.g., a list or array) containing the object IDs present in the web page, and the information provider ID of the information provider who owns those IDs (also present in the web page) to the network 116. The applet puts this vector together into a string, referred to as "object data," and then preferably creates an HTTP message to communicate this information to the network. A schematic example is as follows:

```
POST / HTTP/1.1\r\n
Content-Length: <length of object data> \r\n
\r\n
<object data>
```

where <object data> is <<information provider ID>, <object ID list>>. When the network 116 receives such an HTTP request, it extracts the object data and updates a registry (not shown) indicating which clients have registered for which object IDs from which information providers. In one embodiment, the registry is maintained with a hash

table. A particular client's registrations preferably remain in effect as long as the client is displaying the associated web page 114 and the connection with the network 116 is open. Alternatively, a client 112 may maintain the connection with the network 116 after the client is no longer displaying the web page. In this latter embodiment, the client 112 preferably indicates to the network 116 that it is no longer displaying the objects with which it is registered, in order to reduce bandwidth consumption.

As will be appreciated, any number of clients can register for the same object, and a client can register for as many different objects as it chooses. This registration process creates an efficient many-to-many mapping of objects to clients (and by inference of information, a many-to-many mapping of information providers to clients), without burdening the information provider's own server with maintaining and continuously updating these mappings.

Once it has registered its object IDs, the applet preferably waits until it receives a message from the network 116 in the form of HTTP data. The message will contain an object ID and a new value for the identified object. Upon receipt of the message, the applet preferably extracts the object data contained within. In some cases, the object data is raw JavaScript code. In other cases, the object data is a change to the property of the object having a particular object ID. In this latter case, the applet preferably generates JavaScript code to accomplish this change. In either case, the applet preferably uses LiveConnect to execute the JavaScript code on the client's browser. The JavaScript code, in turn, causes a change in the object in the web page displayed on the display device of the client 112. As more fully explained below, the server 110 may send out updates with any frequency, without relying on the client(s) to request updated information. When a

browser loads a new web page, the applet stops running. This causes the client's connection with the network 116 to close. If the new page contains an instruction, such as described above, to connect to the network, then a new connection is established, and the above process is repeated.

The information provider uses the server 110 or another computer system to connect to the network 116. When the information provider wants to modify an object on the web page 114, the information provider provides the network 116 with a message containing the information provider ID, the object ID for the object and the new value for the object. Alternatively, another entity acting in tandem with the information provider can provide the messages to the network 116. For example, in one embodiment a third-party content provider, such as an entity dedicated to providing sports scores or stock quotes, provides object IDs associated with live content to the information provider. The information provider distributes the third-party object IDs to the information provider's clients. The third-party content providers then communicate directly with the network 116 and send messages updating the objects initially provided to the clients 112 by the information provider.

In one embodiment, a message contains only a single object ID and a new value for the identified object. In an alternative embodiment, a message contains multiple object IDs and their corresponding updated values. In this latter embodiment, the message may have an associated "Batch ID" that identifies the message as having multiple object IDs and values. Preferably, an information provider can include a batch ID in a web page in the same manner as including an object ID. Likewise, a client can preferably register an interest in a batch ID with the network 116 in the same manner as

an object ID. Alternatively, separate procedures can be established for registering batch messages.

As stated above, a message in one embodiment contains an object ID and a new value for the identified object. The new value is essentially a change to a property of the object. For example, in the Document Object Model, many elements of a web document have a property called "innerText," which contains the text displayed by that element. Changing the innerText property of an element causes the displayed text to change. If a web page contains a table having an entry containing a team's baseball game score and marked with the object ID "Bang$homeScoreID," the table entry can be set to "2" by sending a message to this object ID setting the innerText property to the table entry to "2."

More advanced messages can perform any DHTML effect supported by a client's browser. For example, a message routed to a particular object ID may contain instructions for changing the properties of other objects on the page. In a sense, the information provider or third party sending the message has as much control over what appears on the client browser as does any script that the client downloads via the Web. If the browser supports dynamic HTML, Java, JavaScript, Flash, etc., then a message can perform any effect supported by those tools.

Preferably, the information provider or third party uses an application program interface (API) to send messages to the network 116. Preferably, the information provider or third party uses one or more of the tools described below to access the API.

In one embodiment, the information provider or third party executes a program called a "data pump" that reads an XML file containing one or more object IDs and the

new values for the identified objects. The data pump preferably reads the XML file at regular intervals and sends any messages representing changed data to the network 116 through the API. In another embodiment, the data pump polls the file system on the server 110 or another computer system to determine whether the XML file has changed, and then reads the file in response to a change. In yet another embodiment, the data pump is automatically executed in response to a modification to the XML file.

Another embodiment of the present invention uses a "director console" program that presents a user with a graphical interface displaying the contents of a web page. The user uses a computer mouse, keyboard, or other input device to directly edit the web page. For example, the user can use the director console to edit textual data, images, and/or any other objects on the web page. After editing, the user uses a "send update" button to cause the director console to send messages for changed objects to the network 116 via the API. Since an entire web page can have one object ID, a user can use the director console to change the entire web page associated with the object ID.

In another embodiment, the information provider or third party uses a content management system to update web page information on the server 110. For example, if the information provider's web page provides baseball game scores, the information provider may rely on a content management system to update its web pages with new scores received from the third party data provider. In this embodiment, the information provider preferably uses a tool that integrates with or otherwise works in tandem with the content management system to send messages for changed objects to the network 116 through the API at the same time that the content management system updates the web

page on the server 110. In this manner, the web pages at the server 110 and the web pages at clients are updated almost simultaneously.

For example, the tools described above, in one embodiment, generate the following code to generate a message updating a score to "2":

```
LiveObject score = new LiveObject("homeScoreID");
score.setProperty("innerText", "2").
```

This code creates a message setting the innerText attribute of the object having object ID "homeScoreID" to "2."

The network 116, in turn, routes each message received from an information provider or third party to all of the clients that have registered an interest in the object ID referenced by the message. As described above, an applet at the client 112 receives and processes the messages and updates the objects displayed on the web page. As a result, the network 116 allows objects on web pages to be dynamically updated. Preferably, this updating happens quickly enough to be considered "real-time."

The present invention, as described above, performs object-based routing. Messages provided by the information provider to the network 116 are not routed to the clients 112 based on a specified destination, more specifically, are not routed based on the IP address of the client, as in conventional Web systems. Instead, the messages are routed in the network based on object registrations stored by the network 116. Thus, the messages are routed to a dynamically changing cross-section of clients, since the clients are dynamically registering for objects, receiving some number of updates to the objects that are being viewed, and then closing their connections to the network after some arbitrary period of time. In addition, it is not necessary for the client 112 to have any a

priori knowledge of object IDs – it "discovers" which IDs it should register when it receives a web page from the server 110. Equally important, the present invention allows information providers providing web pages from web servers to dynamically update content on web pages without requiring each and every client to re-request the content, and without requiring the server to maintain a connection with each such client. In this manner, significantly more clients can receive updated content from a given information provider's server than could possibly be directly connected the server via a conventional TCP/IP connection.

As described above, content on the Web is conceptualized and organized by "object" instead of by web page. Therefore, the present invention essentially abstracts web page content away from the HTML code and other components that typically form the web page. This abstraction allows an information provider to update an object without concern for the location, display format, or other specifics of the web page.

For example, the information provider can supply a web page to a first client that includes sports scores for all teams in the National Football League. Similarly, the information provider can supply a second web page to a second client that includes sports scores for all professional teams in the San Francisco Bay area. Although both pages could be completely different in terms of layout, formatting, etc., both pages will contain object IDs identifying scores for games involving the two professional football teams in the Bay Area: the San Francisco 49ers and the Oakland Raiders. The information provider updates one of the scores on both pages (or any number of pages containing these scores) by simply sending a message containing the new score and the object ID corresponding to that score to the network 116. Any client viewing a page containing the

score (regardless of the particular page layout and construction) will receive the updated scores. Thus, the present invention reduces the server's overhead since it does not have to maintain multiple instances of the same object in multiple different pages.

FIG. 2 is a high-level block diagram illustrating a network architecture 200 for providing the functionality described above with respect to FIG. 1 according to an embodiment of the present invention. Briefly, FIG. 2 shows multiple information providers 210 (labeled "customer") each communicating with a processing cluster 212. A global load balancer 214 preferably directs a given information provider 210 to a given cluster 212. In addition, a cluster load balancer 216 within each cluster 212 preferably directs each information provider 210 to a particular gateway 218 of the cluster. The gateway 218, in turn, directs messages received from the information provider to one or more nodes 220 in the same cluster 212 or in other clusters. Clients 222 communicate with a given node 220. A client-side global load balancer 224 preferably directs a given client 222 to a given cluster 212. Likewise, a client-side cluster load balancer 226 preferably directs the client 222 to a particular node 222 within the cluster 212. In addition, at least one applet server 228 in each cluster 212 preferably provides the clients directed to that cluster 212 with the Java applet for identifying and registering for object IDs.

Certain components are not illustrated in FIG. 2. For example, components dedicated to security, administering, and monitoring the network 200 are not shown. Likewise, components for performing billing and accounting are not shown. It will be understood that a typical embodiment of the present invention will have such components.

Turning now to the individual components illustrated in FIG. 2, the clusters 212 preferably are distributed geographically in order to more efficiently serve a larger total geographic area. In addition, geographically distributed clusters 212 provide a greater degree of fault tolerance in case there is a power interruption or natural disaster in one area. Although not illustrated in FIG. 2, one or more of the components within each cluster 212 may have one or more backup systems to provide additional fault tolerance.

The global load balancer 214 preferably ensures that load is distributed among the clusters 212 according to a predetermined heuristic. As is known in the art, the global load balancer 214 preferably directs an entity, such as a server or a client, that needs to interact with the network 116 to one of the clusters 212. For example, the load may be distributed evenly among the clusters 212, or a more powerful cluster may be distributed a majority of the load. The client-side global load balancer 224 preferably performs the same function with respect to clients that request access to nodes 220 or applet servers 228 within the clusters 212. In one embodiment, the global load balancer 214 and client-side global load balancer 224 are the same.

Similarly, in one embodiment the cluster load balancer 216 and client-side cluster load balancer 226 are the same. The load balancer 216, 226 initially determines whether an incoming request is from an information provider 210 seeking a connection to a gateway, a client 222 seeking a connection to a node 220, or from a client seeking a connection to an applet server 228. Then, the load balancer 216, 226 distributes the connection requests among the nodes 220, applet servers 228, and/or gateways 218 within the cluster 212. Preferably, the load balancing for the nodes 220 and applet server 228 are performed by the same component since, for security reasons, most client browsers

only permit an applet to transparently connect (i.e., without user intervention) to the location from which the applet was downloaded.

Alternative embodiments of the present invention may combine the global and cluster load balancers and/or incorporate the functionality of the load balancers into different components within or outside of the clusters 212. In addition, alternative embodiments may not use one or more of the load balancers. For example, if each cluster serves a geographic region, it may not be necessary to utilize the global load balancer.

The gateways 218 preferably receive messages from information providers and direct the messages to the appropriate nodes 220. In one embodiment, each gateway 218 maintains a persistent TCP connection to every node 220. Therefore, although a gateway 218 is located inside a cluster 212 and receives connections via the cluster load balancer 216, the gateway's scope spans the entire network 116. In an alternative embodiment of the present invention, each gateway 218 maintains a persistent TCP connection to all nodes 220 in the same cluster 212 and at least one connection to each of the other gateways in the network. Although this alternative embodiment is less fault tolerant than the previously-described embodiment, it is more efficient because it vastly reduces the number of connections maintained by each gateway 218. In another alternative embodiment, each cluster also includes a gatekeeper (not shown in FIG. 2) that maintains connections with the gateways 218 of other clusters 212. A gateway 218 forward messages to the gatekeeper, which then distributes the messages to the gateways of other clusters 212.

Since a gateway 218 does not control the rate at which information providers 210 send messages, it is possible for the gateway to receive messages faster than it can

process them (i.e. send them to the nodes). Therefore, each gateway 218 preferably maintains a queue of messages that have been received but not yet processed in order to avoid losing messages. In one embodiment, the gateway drops messages if the queue becomes too long. In addition, one embodiment assigns priorities to certain messages or certain information providers 210 to help determine which messages to drop.

The nodes 220 preferably transmit messages received from the gateways 218 to the clients 222 that have registered an interest in the object IDs identified by the messages. A node 220 preferably maintains a table or other data structure containing the object IDs registered by clients 222 connected to it. In a preferred embodiment of the present invention, the table is a hash table that associates a (Information provider ID, Object ID) pair with a linked list containing one entry for each client 222 that has registered that object ID from that information provider 210. Each entry in the linked list preferably contains a pointer to a socket representing the connection to the given client 222. The hash table entries are not static. Accordingly, the node 220 preferably adds to the table every time a new client registers interests in some objects and removes a client's entries when the client disconnects.

When a node 220 receives a message from a gateway 218, the node preferably extracts the information provider ID and object ID (or batch ID) from the message and looks these IDs up in the table. The node 220 then transmits the message to those clients in the list. Preferably, the node 220 transmits the message by wrapping it in an HTTP header.

Since a node 220 does not control the rate at which gateways 218 send messages, it is possible for the node to receive messages faster than it can process them (i.e. send

them to the clients 222). Therefore, each node 220 preferably maintains a queue of messages that have been received but not yet processed in order to avoid losing messages. In one embodiment, the node 220 drops messages if the queue becomes too long. In addition, one embodiment assigns priorities to certain messages or certain information providers 210 and/or clients 222 to help determine which messages to drop.

FIG. 3 is a high-level block diagram of a computer system 300 for use as a server 110, client 222, and/or for one or more of the various components illustrated within the network 116 according to one embodiment of the present invention. Illustrated are at least one processor 302 coupled to a bus 304. Also coupled to the bus 304 are a memory 306, a storage device 308, a keyboard 310, a graphics adapter 312, a pointing device 314, and a network adapter 316. A display 318 is coupled to the graphics adapter 312.

The at least one processor 302 may be any specific or general-purpose processor such as an INTEL x86 or POWERPC-compatible central processing unit (CPU). The storage device 308 may be any device capable of holding large amounts of data, like a hard drive, compact disk read-only memory (CD-ROM), DVD, or some other form of fixed or removable storage device. The memory 306 holds instructions and data used by the processor 302. The pointing device 314 may be a mouse, track ball, light pen, touch-sensitive display, or other type of pointing device and is used in combination with the keyboard 310 to input data into the computer system 300. The network adapter 316 couples the computer system 300 to a network such as the Internet.

Program modules 322 for providing the functionality attributed to the server 110, client 222, and/or for one or more of the various components illustrated within the network 116 are preferably stored on the storage device 308, loaded into the memory 306,

and executed by the processor 302. Alternatively, hardware or software modules may be stored elsewhere within the computer system 300. As used herein, the term "module" refers to computer program logic and/or any hardware or circuitry utilized to provide the functionality attributed to the modules. The types of hardware and software within the computer system 300 may vary depending upon how the computer system is utilized. For example, a computer system used as a node 220 is likely to have greater processing power and storage capacity than a computer system used as a client 222. In addition, the node computer system may lack certain components, such as a display 318 or graphics adapter.

The above description is included to illustrate the operation of the preferred embodiments and is not meant to limit the scope of the invention. From the above discussion, many variations will be apparent to one skilled in the relevant art that would yet be encompassed by the spirit and scope of the invention.

22726/05892/SF/5043974.3

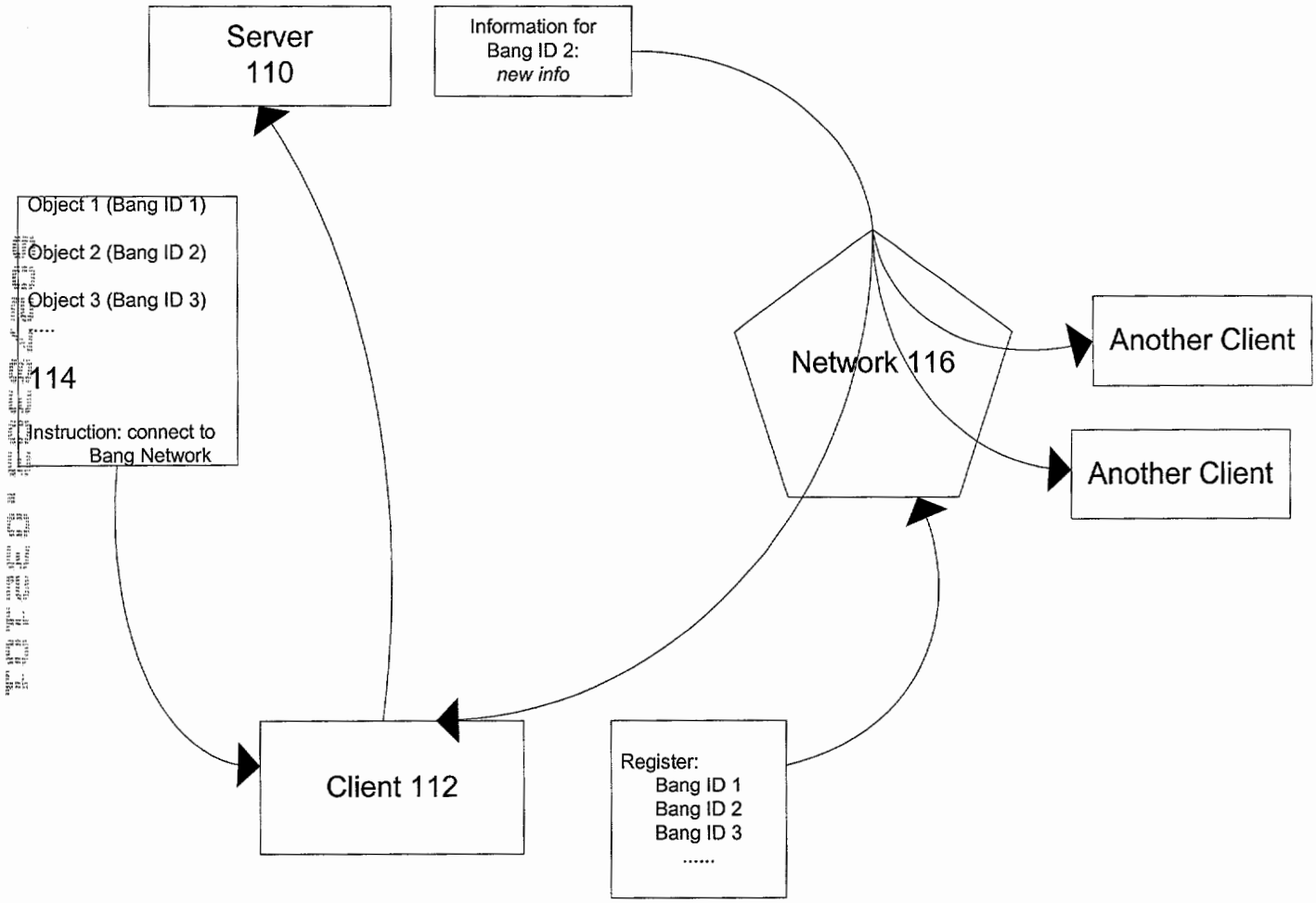
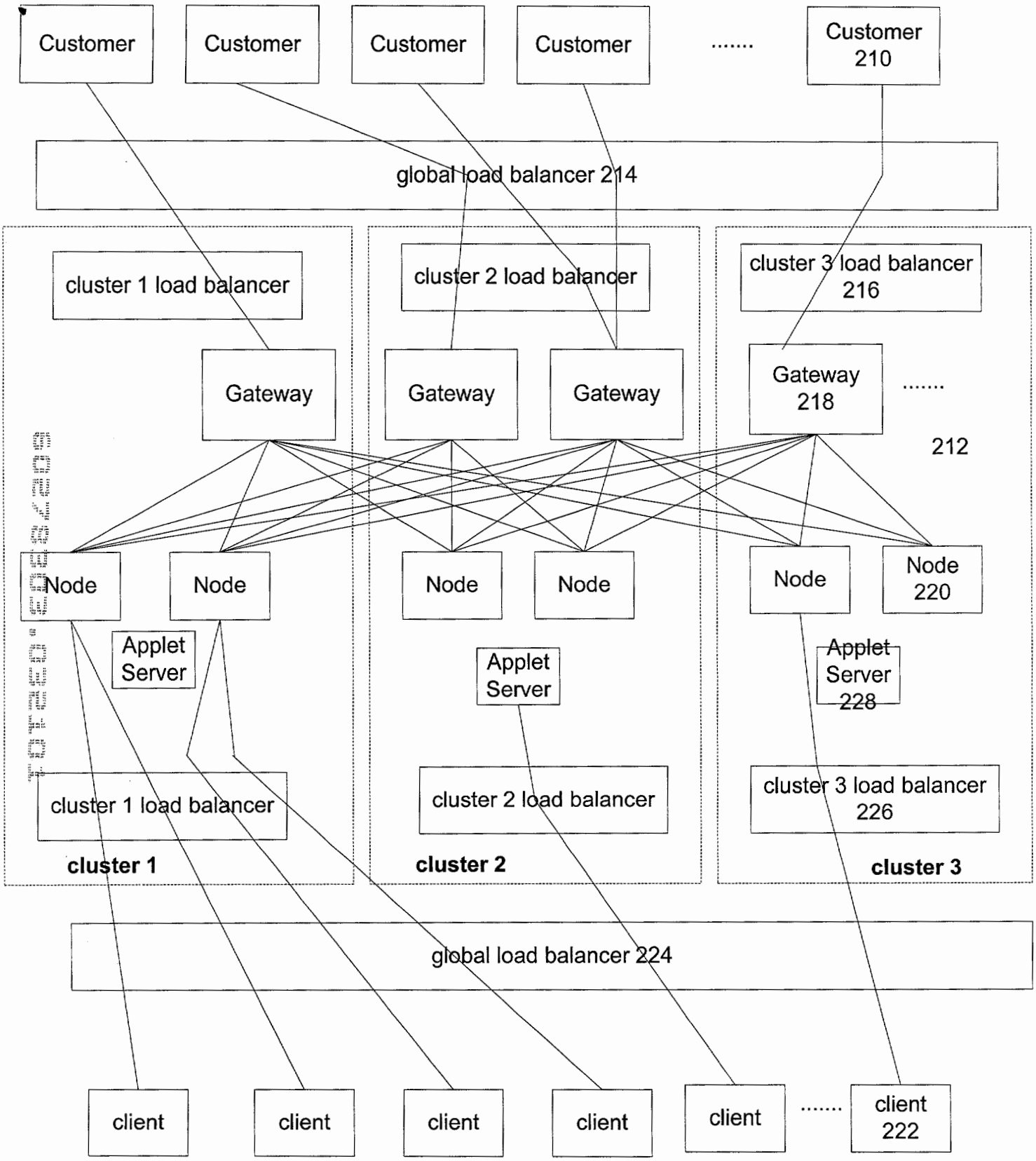

Server
110
Information for
Bang ID 2:
new info
Object 1 (Bang ID 1)
Object 2 (Bang ID 2)
Object 3 (Bang ID 3)
.....
114
Instruction: connect to
Bang Network
Network 116
Another Client
Another Client
Client 112
Register:
Bang ID 1
Bang ID 2
Bang ID 3
.....

FIG. 1


Customer
Customer
Customer
Customer
.....
Customer 210
global load balancer 214
cluster 1 load balancer
cluster 2 load balancer
cluster 3 load balancer 216
Gateway
Gateway
Gateway
Gateway 218
.....
212
Node
Node
Node
Node
Node
Node 220
Applet Server
Applet Server
Applet Server 228
cluster 1 load balancer
cluster 2 load balancer
cluster 3 load balancer 226
cluster 1
cluster 2
cluster 3
global load balancer 224
client
client
client
client
client
.....
client 222

200

FIG. 2

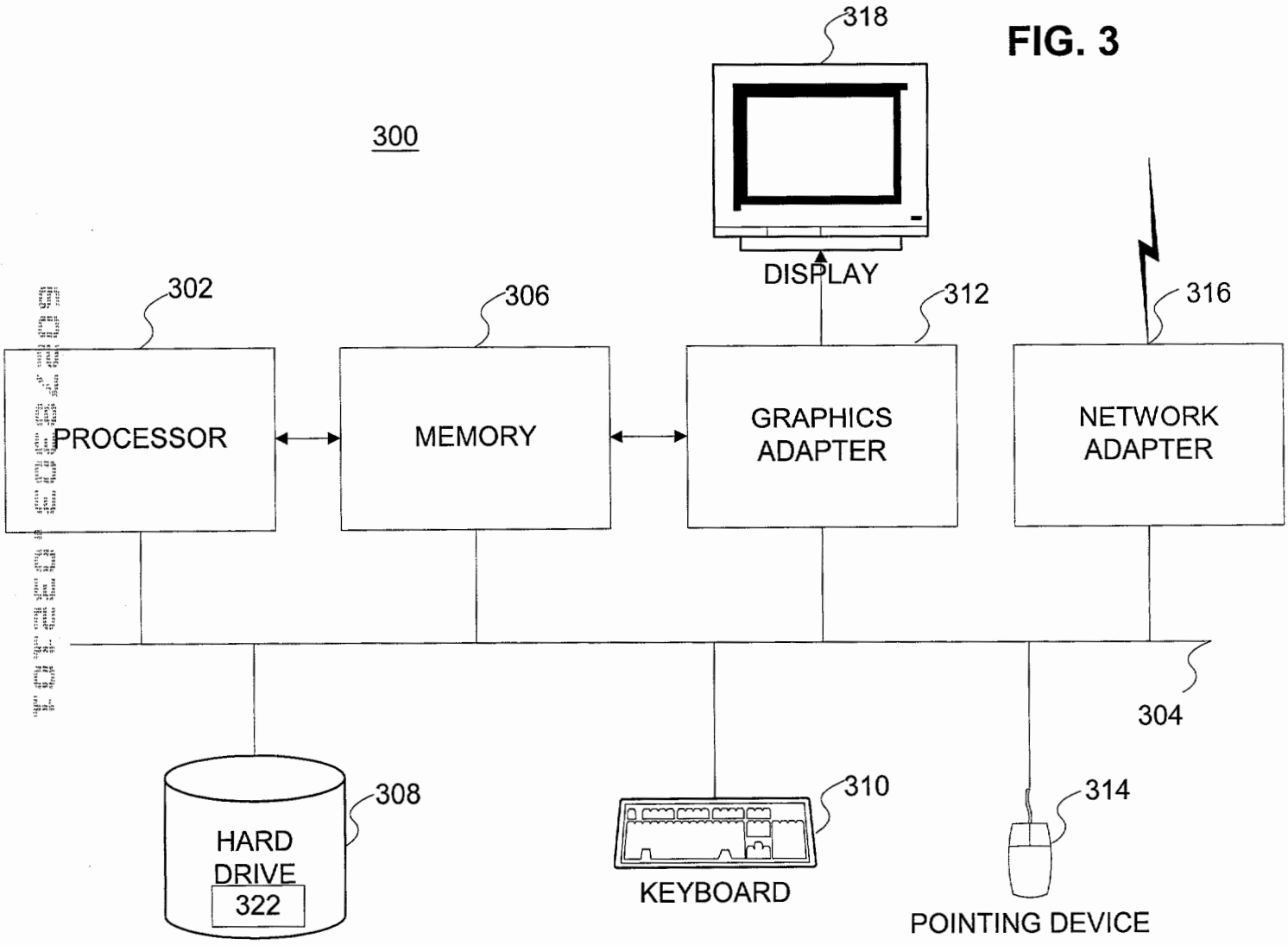


FIG. 3
300
318
DISPLAY
302
PROCESSOR
306
MEMORY
312
GRAPHICS ADAPTER
316
NETWORK ADAPTER
304
308
HARD DRIVE
322
310
KEYBOARD
314
POINTING DEVICE