Title: CLIENT SERVER NETWORK WITH REPLICATED APPLICATION STATE

Abstract: The invention relates to a data processing system (1) comprising: - a first processing device (2) comprising a first processor arranged to load a first operating system (40) and arranged to monitor changes of at least a first object, and - a second processing device (4) comprising a second processor arranged to load a second operating system (180) and arranged to monitor changes of at least a second object, wherein said first processing device (2) is arranged to send a first object change message to said second processing device (4), when a change of said first object is initiated at said first processing device (2), and wherein said second processing device (4) is arranged to send a second object change message to said first processing device (2), when a change of said second object is initiated at said second processing device (4), and wherein said first and said second processing device (2, 4) are arranged to process changes of said first and second objects so as to copy said first object to said second object after a change of said first object, and so as to copy said second object to said first object after a change of said second object.
Client server network with replicated application state

FIELD
5 The present invention relates to data processing systems and more particular to client server networks.

BACKGROUND
10 Nowadays, particular data processing systems are used wherein a user installs a set-top box (STB) in order to use applications from a remote application server. A set-top box is a device that for example enables a television set to become a user interface to the Internet and also enables a television set to receive and decode digital television (DTV) broadcasts. DTV set-top boxes are sometimes called receivers. A set-top box is necessary to television viewers who wish to use their current analog television sets to receive digital broadcasts. Recently, hybrid set-top boxes are available that support digital video broadcasting (DVB) and Internet Protocol (IP)-based video. Such devices allow users to view digital cable programming as well as videos from the Internet or a local IP network. The IP set-top box is shaping up as a critical element in the battle between cable operators and Telco-based broadband providers to provide customers with the all-important "triple-play" of voice, data and video.
15 Typically, an IP STB provides a kind of web browser plus additional functions such as a remote control interface, and a video player. Actually such an IP STB only differs from a PC having a web browser in the fact that:
   - the screen is replaced by a TV,
   - keyboard and mouse are replaced by a remote control,
20   - hardware is arranged in a relatively small box,
   - very often no hard disk is present in the IP STB.

Web pages in fact are interactive in a limited way, i.e. the World Wide Web is designed using a "client pull" technology. This means that the "client" needs to initiate actions and it is not possible to directly communicate with the client from the "server" side. To solve this problem, client side scripting, such as the well known JavaScript is used. However, the JavaScript code is not part of an application running on the server side and as a consequence does not "know" the actual state of the application. Therefore such a "client pull" technology is not very suitable for more advanced interactive
applications.

**SUMMARY**

It is desirable to provide a client server network architecture which is more suitable for advanced interactive applications.

According to an aspect of the invention, there is provided data processing system comprising:

- a first processing device comprising a first processor arranged to load a first operating system and arranged to monitor changes of at least a first object, and
- a second processing device comprising a second processor arranged to load a second operating system and arranged to monitor changes of at least a second object,

wherein the first processing device is arranged to send a first object change message to the second processing device, when a change of the first object is initiated at the first processing device, and wherein the second processing device is arranged to send a second object change message to the first processing device, when a change of the second object is initiated at the second processing device,

and wherein the first and the second processing device are arranged to process changes of the first and second objects so as to copy the first object to the second object after a change of the first object, and so as to copy the second object to the first object after a change of the second object.

By sharing objects used by two or more devices, it is possible to design a multifunctional client server environment with real bi-directional communication (i.e. communication that is initiates from both sides)

The first processor may be arranged to load a first virtual machine (VM) on the first operating system, the first virtual machine being arranged to create the first object change message, and to send the first object change messages to the second processing device.

Implementing the invention in a VM, obviates the need to adjust the existing OS, only the VM needs to be programmed.

The second processor may be arranged to load a second virtual machine on the second operating system, the second virtual machine being arranged to create the second object change message, and to send the second object change messages to the first processing device.

The first processing device may be arranged to execute an application, and the first processor may be arranged to translate an assignment made by the application into the first object change message. In this way the content of an object can be transparently replicated to other data processing
devices without special actions from the applications.

The first processing device may be an application server arranged to execute a plurality of applications.

The first and the second processing device may be arranged to communicate using an internet protocol. In this way the invention can be implemented in a data processing system that can function over existing communication networks.

The internet protocol is arranged to sent messages via the internet, the messages comprising object change information, and identification information of all devices that share the object involved. This will save network bandwidth, because only updates are transmitted via the protocol.

The first and the second processing device may be arranged to communicate with a wide area network protocol. In this way the data processing system according to this embodiment can function on future communication networks.

The data processing system described above may comprise at least a first storage device arranged to communicate with the first processing device and arranged to store the first object.

Separate storage node makes the system more redundant, in case the application server experiences a critical failure, like a power failure, no data is lost nor any application status information, since this status info is stored on the storage device.

The data processing system may comprise at least a third processing device arranged to, at least in use, execute an application that is identical to the application on the first processing device, the third processing device being arranged to monitor at least a third object, wherein the first and the third processing device are arranged to send and receive data to and from the storage device so as to copy the first object to the third object and visa versa after a change of the first or the third object respectively. In this way it is possible that one instance of the application is loaded on several application servers at the same time. One server may go down without the user even noticing this.

The first processing device may comprise an object validator arranged to check if the first object is valid or not, and wherein the first processing device is arranged to load an update for the first object from the storage device only in the situation where the first object is invalid.

The application server stores states (i.e. objects) of an application in its cache, the storage device knows up-to-date status which is sent to application server if object is invalid. The object only needs to be sent to application server if status is invalid, so there is minimal communication between database and the application server.

The data processing system described above may comprise two or more storage devices
arranged to communicate with the first processing device and arranged to store the first object.
In this way several nodes contain the same objects that are exact copies. In that case the second and third nodes function as backups for the first storage nodes.

Data processing system according to any of the preceding claims, wherein the first processing device is arranged to execute a desktop application. Alternatively it may be arranged to execute a video application. It also may be arranged to execute a domotics application.

The second processing device may be arranged to communicate a format of the second object to the first processing device. In this way the first processing device knows the exact format of the object used by the second processing device. This makes it possible that applications can be used to universally control devices of a certain kind, with differences between the exact implementations of those devices (i.e. differences in object formats). A smart application has the ability to adapt to these differences, so the same application can be used to control a plurality of devices of the same kind, but from a different manufacturer.

The invention also relates to the use of a data processing system as described above. Finally the invention relates to a method according to claim 19.

BRIEF DESCRIPTION OF THE DRAWINGS

Embodiments of the invention will now be described, by way of example only, with reference to the accompanying schematic drawings in which corresponding reference symbols indicate corresponding parts, and in which:

- Figure 1 schematically depicts a data processing system according to an embodiment of the invention;
- Figure 2 schematically depicts some functional building blocks of a processor in the first processing device;
- Figure 3 schematically depicts the inner workings of the virtual machine of the first processing device;
- Figure 4 schematically depicts the building block of a REP handler inside the virtual machine of figure 3;
- Figure 5 schematically depicts a REP Router and its interaction with two devices;
- Figure 6 schematically depicts some functional building blocks of a processor in the second processing device;
- Figure 7 schematically depicts the inner workings of the virtual machine of the second
processing device;
- Figure 8 schematically depicts the building blocks of a RIP message handler;
- Figure 9 schematically shows the functional structure of the RIP proxies;
- Figure 10 schematically depicts the Central Object Store.

DETAILED DESCRIPTION

Figure 1 schematically depicts a data processing system 1 according to an embodiment of the invention, comprising a first processing device 2 and a second processing device 4. The first processing device 2 comprises a processor (not shown) arranged to process instructions and a memory (not shown) arranged to store data and instructions. The first processing device 2 may for example be an application server 2 arranged execute a specific application, such as a word processor application. According to an embodiment, the processor of the device 2 is arranged to monitor at least a first object comprising data and/or instructions of the application. The object may be stored in the memory of the application server 2, such as a local RAM, but it may also be stored in one or more storage node as will be explained below in more detail.

In an embodiment, the application is programmed using an object oriented programming language. In that case, the first object may comprise data of a specific object used by the application. If the application is not object oriented, the first object may for example comprise all data used by the application. Besides data, the first object may also comprise instructions related to the data in the object.

The second processing device 4 is arranged to store a second object. Below, the second processing device 4 will also be referred to as client device 4. In an embodiment, the client device 4 is a STB 4. The STB 4 comprises a memory to store the second object. The STB 4 may comprise a processor arranged to load an operating system. In the embodiment of figure 1, the STB 4 is connected to the application server 2 via an IP-based communication network 6, such as the Internet. Furthermore, the STB 4 is connected to a TV 8 and to an I/O device 10, such as a keyboard 10. The application server 2 is connected to the communication network 6 via a load balancer 12 and some proxies 14, 16 and some routers 18, 20 which will be discussed in more detail below.

In an embodiment, the data processing system 1 also comprises a third processing device 22, that is arranged to execute an application which is the same as the application executed on the application server 2. Both application servers 2, 22 may be connected to a Central Object Store (COS)
25, see figure 1. The COS 25 is arranged to store the first object into for example a object oriented database, as is known to the skilled person.

According to an embodiment, particular objects are replicated (i.e. copied) from the application server 2 to the client device 4 and vice versa. The objects that are replicated are referred to as 'shared objects'. According to an embodiment, the updates for the shared objects are sent via an IP-based network using a specific protocol named Remote Event Protocol (REP).

Figure 2 shows schematically an example of the application server 2. The application server 2 comprises an operating system 40 and a virtual machine 42 which, at least in use, is loaded on top of the operating system 40. The virtual machine 42 is the environment where the actual processing of applications takes place. In this example, the application server 2 is arranged to execute two applications X and Y. In figure 2, the application X is loaded twice by different users, resulting in two application instances 44, 46. The application Y is loaded once resulting in application instance 48. Below, the term “application” will be used for both application and application instance unless the specific semantical difference needs to emphasized.

The virtual machine 42 hides the actual operating system 40 from the application 44, 46, 48. The environment the virtual machine 42 offers for the applications 44, 46, 48 is highly platform independent. The specific implementation of the virtual machine 42 and its software components, such as an OS dependent layer 41, is platform dependent because it needs to interact with the operating system 40 and/or hardware it is physically running on.

Now, the inner workings of the virtual machine 42 will be high-lighted with reference to figure 3. The virtual machine 42 partly resembles known virtual machines, such as the Java Virtual Machine, but it differs with known virtual machines in the way that it can interpret objects. Most other virtual machines only execute basic CPU functionality i.e. machine instructions. Unlike known virtual machine implementations which only implement a virtual machine in the form of a simulation of basic machine instructions and are not 'aware' of the objects that are processed, the virtual machine 42 according to this embodiment is capable of monitoring object changes. At the core of the virtual machine 42 is an instruction processor 50 which operates much like an interpreter for high level programming languages. The instruction processor 50 is arranged to process object transitions within the applications 44, 46, 48, but it also executes instructions needed to interface with the OS dependent layer 41, or other possible components connected to the virtual machine 42.

The instruction processor 50 is arranged to perform a standard task and additional tasks with all kinds of "value assignments" and "value requests". In a nowadays computer program, an
assignment to a variable (whether or not part of an object) causes this value to be stored into memory. If this value is used in an instruction later on, this value will be read from the memory and temporary stored in a register. The instructions will then be executed and results of the instructions will be written back into memory if there was an object change caused by the instructions.

The instruction processor 50 is arranged to process those instructions (this is meant by the standard task). The processing consists of parsing the instruction, identifying the objects involved in the instruction and executing the instruction. Once it recognizes 'read' or 'write' request to an object, it will perform an additional task. This additional tasks include replicating object changes to the clients the object is shared with.

Since in object oriented applications all application data is stored in objects (it doesn’t matter whether or not the application is aware of this), an assignment of a new value does mean an object has changed. When the applications 44, 46, 48 use a stored value in a particular instruction, they need to retrieve data from an object, before it can be used in that instruction.

Please note that in traditional applications running on a traditional operating system, all objects are stored into local memory. An assignment of a new value, means the new value will be stored into memory. If the application processes an instruction, involving the stored value of this object, this means the value has to be retrieved from memory. In most cases, the process of storing and retrieving the values is left to the operating system. From the applications point of view, the virtual machine 42 according to the embodiment is completely transparent. No adjustments are needed in the applications.

The virtual machine 42 furthermore comprises an application storage handler 52 and a application storage cache 54 which is arranged to communicate with said instruction processor 50 and the application storage handler 52. The application storage cache 54 is arranged to cache (i.e. store) read requests. The virtual machine 42 also comprises an instruction cache 56 arranged to store instructions from said instruction processor 50. Furthermore the virtual machine 42 comprises a so-called object investigator 58, and two message handlers named ‘REP message handler’ 60 and ‘RIP message handler’ 62. Both REP message handler 60 and RIP message handler 62 communicate with the instruction processor 50 and with external devices, such as the not shown. The REP message handler 60 also receives input from the object investigator 58, see arrow 61.

The application storage handler 52 is arranged to communicate with the central object store 25, see also figure 1. It handles I/O requests and ensures that storage transactions are successfully processed. The application storage handler 52 receives object (updates) from the COS 25, see arrow
53. It may also store an object into the COS 25, see arrow 51. If the application storage handler 52 receives an ‘object invalidation’ message from one of the COS 25, it passes this message to the application storage cache 54, see arrow 55. The application storage cache 54 will then acknowledge the receipt, see arrow 49 and remove the corresponding cached object from its memory because it is no longer valid. In this way, only valid objects are stored in the application storage cache 54. The use of the application storage cache 54 greatly enhances the processing speed of the virtual machine 42.

Objects within the application storage cache 54 become invalid, once they have been updated outside the scope of the virtual machine 42. For example if the object has been updated by another virtual machine on another application server. Object Invalidation requests are sent as REP message by the virtual machine that has updated a certain object. Those REP messages are sent to the virtual machines of all processing devices that use the same object. In this way, all devices can be notified of an object being invalid. If an object is invalid, that object needs to be retrieved from the COS 25.

The instruction processor 50 passes all assignment made by the application 44, 46, 48, to the object investigator 58, see arrow 71. The investigator will acknowledge the receipt, see arrow 76.

An example of an assignment is:

\[
\text{Object}_1\.\text{property}_20 : = \text{value}_1;
\]

This assignment results in a transition of Object_1 from Object_1old to Object_1new. The object investigator 58 is arranged to compare old objects involved in an assignment as they existed before the transition, with new objects after the transition. For the objects for which the object investigator 58 finds a change, an update request is sent to the application storage handler 52, see arrow 64. The update request 64 only contains the update, so in the example mentioned above the update request contains: Object_1\.property_20 := value_1. However, in some situations such as the creation of a new object, the update request contains the whole object, for example:

\[
\text{Object}_2 : = \text{NewObject}.
\]

If the object which is changed, is a shared object, the object investigator 58 will also send a message to the REP message handler 60, containing the object update, see arrow 61. The REP message handler 60 will forward this message to all client devices 4 this object is shared with, see arrow 66. In an embodiment, a shared object contains an “index” which is a list of devices to which
that particular object is shared to and what permissions those devices have on this object. This index is part of this object, and will only be used by applications that are designed for sharing objects with multiple client devices. The REP message handler 60 is also arranged to receive incoming REP requests (i.e. messages containing a request) from other devices, see arrow 68. The REP message handler 60 sends the incoming REP requests to the instruction processor 50, see arrow 70. The instruction processor 50 acts upon this request by executing a request described in the REP message received. Most of the time, a request will include an object update. But there may also be several other requests that can be contained within a REP message, for example a request to create a new object or to delete an existing object.

The instruction processor 50 may send [correct??] and receive an object (or object update) to/from the application storage cache 54, see arrow 72, as well as to/from the REP message handler 60, see arrow 70. In both cases the object (update) may comprise data and/or instructions associated with the data. The instructions of the object are stored into the instruction cache 56 by the instruction processor 50, see arrow 74. These instructions are then loaded from the instruction cache 56, see arrow 73 and executed by the instruction processor 50 in order to update the state of the application 44, 46, 48.

This sequence (i.e. module 52->54->50->56->50) is repeated for every object update. After the instruction cache 56 has been filled, the instruction processor 50 starts executing the instructions contained in the instruction cache 56. The execution of those instructions is actually the heart of the execution of the application 44, 46, 48. The execution of the instructions associated with object updates may result in new object changes, which in case of shared objects, will be updated on the client device 4 to reflect the new state.

The RIP handler 62 is arranged to receive and send RIP messages. It receives requests to create a RIP message from the instruction processor 50 which are forwarded to the RIP proxy, which will then pass this message to the destination for further processing.

Figure 4 schematically shows the building blocks of the REP message handler 60 of the virtual machine 42 of figure 3 together with one of the REP routers 18 shown in figure 1. The REP router 18 sends a REP message to an REP message I/O module 90 comprises in the REP message handler 60 via the REP protocol, see arrow 91. The REP router 18 also receives REP messages from the REP message I/O module 90 via the REP protocol, see arrow 92. The REP message I/O module 90 places incoming REP messages into an incoming REP queue 94, see arrow 95. The REP message I/O module 90 gets the top message from an outgoing REP queue 98, see arrow 96. An object decruncher 100 gets
serialized REP message from the incoming REP queue 94, see arrow 101 and converts it into an object recognized by the application. An object cruncher 10 gets a “system object”, serializes it into a REP message and puts it into the outgoing REP queue 98, see arrow 102. The object decruncher 100 unserializes REP messages and forwards the resulting objects to a REP Call Dispatcher 106, see arrow 107. If the unserialization fails, an error message is sent to the REP call generator 110, see arrow 121. The REP Call Dispatcher 106 sends authentication requests 109 to a REP authenticator 108. The REP authenticator 108 processes request resulting in the following replies:

- success: the processing can continue.
- failure: the processing cannot continue

If the REP authenticator 108 reports a failure, a failure REP message will be generated and forwarded to a REP call generator 110, see arrow 111. The REP Call Dispatcher 106 looks into a REP Call Registry 112 which is arranged to register “calls” or “events” that have to be executed once a certain element of an object changes (e.g. a “keyboard.keydown” property of an object named “keyboard”), see arrows 113, 114. If the defined call is not registered in the REP Call Registry 112, the REP Call dispatcher 106 will abort the processing. A failure message, see arrow 115, stating that the “REP call” cannot be processed because no call was registered for it will be injected into the outgoing REP queue 98 via the REP call generator 110 and the object cruncher 104, see arrow 116. If the particular call defined in the REP message is registered by the application concerned, the REP Call Dispatcher 106 will call the corresponding function call, see arrow 117 and will pass the corresponding object to this call. The application 44, 46, 48, which is loaded on the instruction processor 50, will report the result of this call back to the REP Call Dispatcher 106, see arrow 70:

- success: the call was succesfull.
- failure, the call was not successful, additionally, the REP call generator 110 will be sent a request, see arrow 115, to create a REP message to the original sender, containing an error code, stating why the call failed.

The application 44, 46, 48 generates a so-called “REP event” and will pass this to the REP call generator 110, see arrow 120. The REP call generator 110 will report to the application 44, 46, 48:

- success: the message has been successfully injected into the outgoing REP queue.
- failure: the message injection failed, additionally an error code will be passed to the application 44, 46, 48, stating a reason why the injection failed.

The REP call generator 110 will then pass the object change through the object cruncher 104,
serializing the object changes into a REP message. The REP message is then sent to the REP router 18, 20, see arrow 92.

Figure 5 schematically depicts an example of the REP router 18, 20, see also figure 1. The REP router 18, 20 comprises several building blocks. In figure 5, an example is shown wherein a REP message is sent from the application server 2, 22 to the client device 4. The REP handler 60 of the virtual machine 42, see also figure 3, of the application server 2, 22 sends a REP message, see arrow 140, to a REP message injector thread 142 comprised in the REP router 18, 20. The REP message injector thread 142 will pass the message into a REP message queue 144, see arrow 146. The REP message queue 144 sends the message 146 to a priority scheduler 148, see arrow 150. The priority scheduler 148 is arranged to determine the location in the queue based on a priority flag in the REP message. The priority scheduler 148 will then insert the message into the REP message queue 144, see arrow 152. The REP message queue 144 will pass messages ready for delivery to a REP delivery thread 154, see arrow 156. The REP delivery thread 154 will look into a REP registry database 158, see arrow 160. The REP registry database 158 comprises a list of addresses of REP handlers in the data processing system. The REP registry database 158 determines the physical address of an involved receiving REP handler 164, and sends this to the REP delivery thread 154, see arrow 162. The REP delivery thread 154 will try to deliver the REP message to the receiving REP handler 164. In this example, the receiving REP handler 164 is comprised in the client device 4. If the receiving REP handler 164 generates an error, see arrow 168, the REP message will be sent to a failure processor 170, see arrow 172. The failure processor 170 decides the fate of the REP message:

- If the REP message has been in the REP message queue 144 for too long (i.e. longer than a predetermined timeout threshold), a new REP message is generated with as destination, the original sender of the message (i.e. the application server 2, 22), stating that the REP message has been deleted from the queue because of a timeout.

- If the REP message is before the predefined timeout threshold, the REP message will be injected into the REP message queue 144 again.

Figure 6 schematically depicts an example of the client device 4, see also figure 1. The client device 4 may comprise a STB 4 which can be connected to a TV or it may be for example a processing module of a domestic apparatus, such as a microwave. The client device 4 comprises a processor arranged to load a client operating system 180 and a client virtual machine 182. The client virtual machine contains an OS dependent layer 183 that is arranged to replicate object updates, received by
the client virtual machine 182 into the operating system 180 and to replicate updates from the operating system 180 into the virtual machine 180.

Figure 7 shows the functional building blocks of the client virtual machine 182 according to an embodiment. The client virtual machine 182 comprises an instruction processor 200, an instruction cache 202, an object investigator 204, a RIP message handler 206 and the REP message handler 164 as was already shown in figure 5. The building blocks of the client virtual machine 182 interact the same way as the corresponding building blocks comprised in the virtual machine 42 of the application server 2, see figure 3. Because there is no application loaded on the client device 4, the client virtual machine 182 does not need an application storage handler nor an application storage cache, as were present in the virtual machine 42 of the application server 2, see figure 3.

As was mentioned above, the client device 4 is arranged to store objects. For storing the objects, the client device 4 may comprise a memory such as a RAM or ROM. Instructions contained in the objects are stored in the instruction cache 202. Object data contained in the objects is stored in an object cache, not shown. In an embodiment, the OS dependent layer 183 is arranged to publish certain objects, i.e. make certain objects available for the virtual machine 182 of the client device 4. The OS dependent layer 183 publishes objects that are relevant for the particular client device 4 (e.g. a screen object, a keyboard object, a mouse object)

The application objects can be classified as shared objects so that they are replicated after each update to e.g. the application server 2 by means of the REP protocol. The types of objects that can be shared may be dependent upon the type of client device. The type of objects may vary. One possible object type may be a high level “graphics API”, as is a common option found in most modern client devices. An example is the Windows GUI.

The OS dependent layer 183 also acts upon certain object updates received from the operating system 180 (note that this applies only if the operating system 180 can handle objects itself, because not all common operating systems can do this). The OS dependent layer 183 may also trigger object updates itself, when it for example, receives signals from the operating system 180.

It is noted that the client device 4 may also have an object cache (not shown) in order to avoid non-stop lookups of object data via the REP communication between the client device 4 and the application server 2.

In an embodiment, the application server 2 and the client device 4 communicate using a second protocol referred to as Remote Interface Protocol (RIP). Both the virtual machine 42, see figure 3 and the client virtual machine, see figure 7, may comprise a RIP message handler 62, 206. The building
blocks of the RIP message handler 62, 206 are shown in figure 8. The RIP message handler 62, 20 comprises a RIP message parser 210, an incoming RIP queue 212, an outgoing RIP queue 214 and a application RIP message handler 216. The application RIP message handler 216 is arranged to communicate with RIP message handlers in other devices via the RIP proxies 14, 16, see also figure 1. The incoming RIP queue 212 and the outgoing RIP queue 214 are each arranged to temporarily store RIP messages. The RIP message parser 210 is arranged to communicate with the application 44, 46, 48. The application 44, 46, 48 can create a request to form a new RIP message, this is called a RIP call from application 209, see figure 4. The call is sent to the RIP message parser 210 for further processing. Once a reply has been received from the RIP message parser 210, the message, containing the reply is sent back to the application 44, 46, 48.

In an embodiment, the RIP message handler 62 in the application server 2 is arranged to communicate with the RIP message handler 206 of the client device 4 through one of the RIP proxies 14, 16. Figure 9 schematically shows the functional structure of the RIP proxies 14, 16. The RIP proxies 14, 16 handle all incoming RIP calls from the client devices 4. The RIP proxies 14, 16 act as proxy to the applications on the application servers 2, 22. Each of the RIP proxies 14, 16 comprises a RIP request handler 230, a request authenticator 232 and a RIP application registry 234. RIP message communication may be as follows. The RIP handler 206 at the client device 4 sends a RIP message to the RIP request handler 230, see arrow 236. The RIP handler 230 passes this message to the Request authenticator 232, see arrow 238. The request authenticator 232 validates the message and returns the following status, see arrow 240:
- success: The request could be authenticated and will be authorized for further processing,
- failure: the request could not be authenticated and will be aborted.

Next, the RIP request handler 230 will look up the destination of the RIP request in the RIP application registry 234, see arrow 242. The RIP application registry 234 will return the following result, see arrow 244:
- success: The destination could be found, additionally the physical address is returned to the RIP request handler 230,
- failure: The destination could not be found in the RIP application registry 234.

In the case of success, the RIP message will be forwarded to the destined Application RIP handler 62, see arrow 246. On receipt, the Application RIP handler 62 will report one of the following, see arrow 248:
- success: accompanied with the result of the RIP call,
- failure: the request failed, additionally an error code will be passed.

The RIP request handler 230 will do the following, see arrow 250:

- If there was a failure, report the failure back to the RIP handler 206,
- If there is a result from the Application RIP handler, report this result back.

The RIP protocol consists of a carrier protocol, and may be implemented on layer 3 or 4 of a OSI compliant networking stack (such as IP). In an IP environment, it will act on Layer 4 as protocol implemented over TCP for non-time critical communication and UDP as critical communication. The RIP protocol consists of a header of fixed size, describing the application source ID, the device destination ID and also the device source ID. Additionally, it contains sequence information (A sequence identifier) and payload description information, like the message type (currently defined: new application session, authentication, application payload and error). The payload is usually encrypted using standard encryption techniques. The payload itself can be almost of any protocol, but it has been optimized for carrying time-sensitive data like RDP and ICA sessions. Instead of RIP also other protocols can be used like: XUL, ICA (Citrix), RDP (Microsoft), VNC (RealVNC) or even XWindows. The RIP protocol may be used to communicate display updates via the communication network 6. However, display updates can also be communicated as REP messages to the client devices 4. Contrary to RIP messages, REP messages do not include bitmap data for screen-updates but they describe the screen in "objects".

According to an embodiment, the virtual machine 42 at the application server 2 does not store the objects data in "computer memory", but it assigns this task to the COS 25. Figure 10 shows a schematic overview of the modules of the COS 25 together with the application storage handler 52 of the virtual machine 42 at the application server 2. The COS 25 comprises one or more storage handler nodes 24 and one or more storage nodes 302, 304, 306 and 308. The storage node 24 comprises a transaction handler 314, a transaction cache 312, a request authenticator 314 and an object invalidator 316. In the event of an object request from the application 44, 46, 48, the application storage handler 52 sends a "transaction request" to the central object store, see arrow 320. An object request can be:

- new object: creates a new object in the database
- update object: updates an existing object in the database
- delete object: deletes an existing object in the database
- read object: reads an object from the database
The application storage handler 52 gets a response, see arrow 322, from the transaction handler 310, the possible responses are:
- success: the transaction completed successfully. If the request was of type “read”, the resulting object will also be returned.
- failure: the transaction did not complete, additionally an error number will be passed.

The transaction is passed to the request authenticator 314 by a function call, see arrow 324. The request authenticator 314 sends back one of the following messages, see arrow 326:
- successful: the request is authenticated and authorized to proceed
- denied: the request is denied
- failure: an internal failure happened, additionally an error number will be passed.

If the request is of type “read”, the request will be passed to the transaction cache 312, see arrow 328. The transaction cache 312 will pass one of the following messages back to the transaction handler 310, see arrow 330:
- success: if the transaction can be matched with a ‘Transaction Cache LRU list’ present in the transaction cache 312; also the resulting object will be returned to the transaction handler,
- fail; the object has to be fetched from the storage nodes 302, 304, 306, 308.

The transaction handler 310 will pass the request to one of the connected node fronts 302n, 304n, 306n, 308n. The node front will be determined on the following basis:
- the node front has to be registered into the transaction handler, marking it “alive”.
- the node front with the lowest request/second rate.
- if two nodes have the same load, the first node available will be chosen.

If the request is of type “update”, a response from all the online object invalidators is awaited. The result will be passed back to the transaction handler 310, see arrows 340, 342, 344, 346, the result can be:
- success: accompanied by the resulting object, if the request was of type “read”.
- failure: the request didn’t succeed, an error code will also be passed.

If the request is of type “read”, the object will be read from one of the file system storages 302f, 304f, 306f, 308f. If the request is of type “new” or “update” the update will be saved to all storage nodes 302, 304, 306, 308. By always storing new or updated objects into all the storage nodes, the data between those nodes will stay consistent.

If the request to the node front was of type “update”, an invalidation message will be sent to the object invalidator 316 of the storage handler node 24, see arrows 350, 352, 354, 356. The object invalidator
316 will send a response once the message has been successfully processed. If the message is not received within a time-out frame, the node fronts 302n, 304n, 306n, 308n will pass an error to the transaction handler 310. The object invalidator 316 will remove the object from a Least Recently Used (LRU) list of the transaction cache 312. Once this is successful, the object invalidator 316 will send a message back to the application storage handler 52 via the transaction cache 312 and the transaction handler 310, see arrows 360, 330 and 322.

According to an embodiment, the third processing device 22, referred to as application server 22, is also arranged to communicate with the COS in the same way as the application server 2. The application servers 2, 22 do not store the objects in their own memory. Instead, the objects are stored centrally in the COS. comprises not only the same instance of the application, but also the exact same application state.

In the following, three applications are discussed which can be implemented using the data processing system as described above. These three applications are used as an example and are not intended to limit the scope of the invention by any means.

Example 1: Word processor

In an embodiment of the invention, the first processing device 2 is arranged to execute a word processor. A word processor is a computer software program that is arranged to write a text document by using an automated device like a Personal Computer. The best known Word processor is Microsoft Word, other well-known products are WordPerfect and Star Office Writer. Most modern word processors also implement additional features, like text formatting, font selection, spelling-checker and “What You See is What You Get”. A traditional word processor is a stand-alone application that runs as an instance on a single computer. In a “terminal based” solution, the application runs on a remote server, also as an instance on a single server, only input and output is relayed to a remote client computer.

To better understand the inner workings of a word processor, some components used are discussed. The main feature of a word processor is a large window, featuring a large input field, which displays the text document being worked on. This window also allows manipulation of the contents of this document. Most modern word processors also feature additional input controls like pull-down menu’s to operate the functions of the word processors like text formatting, loading and saving a document,
Most modern word processors use object oriented programming (OOP) to implement the software. Looking from “high altitude” at a word processor, there are just a few objects that define the state of the application. The most important object is that containing the contents of the document you’re working on. This object is referred to as the Document object. Additionally a “view port” is defined, the view port is the piece of the document that is visible to the user. The view port will be a property of the Document object. Additionally, there may be a screen object. The screen object represents the data displayed to the screen. It is noted that in Windows and most other operating systems, the OS provides a framework for such objects, as will be known to the skilled person.

Objects are usually stored in the memory of the computer, usually the RAM memory, because this has a low latency access time. Once the software starts, it will set-up a fresh instance of the document object. If a request comes to open an existing document, the contents of this document will be loaded from the file into the object. Once a user starts typing, the Operating System will produce events for every key stroke. The operating system sends those events to the event handler of the application the keystrokes are destined for. The application receives the events and based on the type of those events, it will decide what to do. For example it will update the contents of the Document Object or adjust the position of the view port if you pushed the “Page down” button for example.

After every event, the application will evaluate if the screen display has to be updated. If the Document Object has been updated, this has implications on the data that is displayed to the user in most of the cases. So this implies that the Screen object has to be updated accordingly, to represent the changes to the Document Object on the screen. Of course, a word processor features a lot more functionality than described above, but it follows all the same guidelines:

- The application receives an event via the operating system. This event can be produced by a click on a mouse, a keystroke on the keyboard or any other input device.

- The application evaluates the implications of the event and manipulates the objects involved accordingly.

- The application inspects if those changes imply a change to the information displayed to the end-user and update this information when needed.

The simple “algorithm” described above is the main-application loop of most modern applications.

The invention keeps the traditional idea of building applications intact. In the word processor embodiment described above, the word processor application can still use its own objects to store its
‘state’ without the need for adjustments. This is not the case with current web-applications, which require an “inside out” implementation. Traditional web-applications don’t have the possibility to directly communicate with the client, resulting in a complicated programming style, almost totally different from the traditional object oriented way of building application. Looking at the resulting code, it almost looks like as if the application was built “inside-out”. This is because instead of using the objects, “browser transactions” need to be processed. In present web-application, there is no real bidirectional communication possible between the client and the server.

According to an embodiment, the objects are not stored in the RAM of the local computer (i.e. the application server), but in the central object store (COS). As was explained above, the application server 2 can share the state of an object with the client device 4. Examples of such shared objects are the screen and input devices like a keyboard. Once an object is marked as “shared”, the state of this object will be replicated between the client device 4 and the COS.

Now the inner workings of this mechanism according to an embodiment will be explained for the word processor application. First, the client device 4 will receive a command via the I/O device 10 connected to the client device 4. The client device 4 will then initiate a request to start a new instance of the word processor. This request will be relayed via a REP message to the application server 2 which is responsible for handling this requests. The application server 2 chosen will then perform the task of setting up an environment for the instance in the central object store 300, 302. Once the application returns into a wait state, control for the execution thread will be handed over to the thread manager of the application. Also, some shared objects will be created by the virtual machine 42 of the application server 2; one will be the screen object and the other is an object referred to as the “Input object”. The input object comprises data and instructions to handle all the input received for the application. If a keystroke or other input event occurs, this will result in a change/update of the input object on the client device 4. The update of this object will be communicated via a REP call to the REP handler 60 at the application server 2. Once the REP message has been received by the REP handler 60, the REP handler 60 will update the object in the Central Object Store accordingly. After this, the update will be passed to the application 44, 46, 48. The application 44, 46, 48 then takes the actions accordingly, like adding the pressed key to the Document Object or changing the font type of a piece of text. During this Document Object update process, the screen will have to be redrawn to match the updated version of the document. The application at the application server 2 manipulates the Screen object accordingly. Since this screen object is a shared object, (the status of) this object will be
replicated to the client device 4. Once the object has been updated in the Central Object Store, the REP handler 60 will send a screen update REP call to the client device 4, also, one or more RIP messages may be generated, containing the screen information. Shared objects are kept synchronous using the REP protocol, but Screen objects are relatively large and therefore a more efficient way of replication is used, via the specialized RIP protocol. It should be noted that another protocol can be used, such as XUL or the upcoming XAML protocol. Also an adapted version of RDP or ICA is possible.

Example 2: Video player

In an embodiment, the first processing device 2 is arranged to execute a video player application. A video player application is a special case. In the implementation of a video player, the most processing has to be performed on the client device 4. But in order to keep control over the video player, the video player makes ingenious use of the fact that the invention provides a "distributed-but-synchronized" object framework.

A very well known example of a traditional video player or, in this case, a media player which happens also to play video is the Windows Media Player. Traditional video players are quite simple programs. They can open video files, or sometimes also network streams and play them on the users screen and audio system. Most video players offer some kind of control functionality, making it possible to navigate through a video. A video player uses a codec to decode the video-data received from disk or network stream. Raw video and audio data needs huge amount of storage space and/or network bandwidth. Most codecs use a way to compress this data, to make storage more efficient and transportation via a network more feasible. A full-featured full-screen video can easily take 100 mbit/s or more of video bandwidth, this makes video-streaming unusable for current broadband networks. Modern codecs can compress video by a factor of 40 with the remaining quality still good enough for consumers. Now first a state of the art video player is discussed, in this case, a video player playing a stream over a broadband connection. Like the Word Processor mentioned above, a modern video player will also feature an "object model". The major objects are:
- The video source, in this case, a network stream.
- The screen output
- A codec object
- A "control object" representing control buttons like "Play", "Fast forward", etc.

Once the application starts, it will initialize the above mentioned objects, tell the video-source
that it is ready to receive the video and it will then start the main application loop which will look like this:
- Check for any new events like a push on a control button.
- In case of a “control event”, take appropriate action like forward the video or stop the playback.
- Get the data from the network source into the “video source” object.
- If the video is still on playback, read the data for next frame from the “video source”.
- Send the data through the codec object for decoding.
- Update the “control object” to reflect the new status like the new frame number.
- Pass the data from the codec to the “screen object” to redraw the screen.

Now, a video player application according to an embodiment is described. For the programmer and the user of the video player according to this embodiment, the implementation only differs from the state of the art in subtle details. The main principle will be the same, but what happens behind the screen is a lot different. First of all, the objects used in the invention will be the same as traditional objects. A major difference is that some of those objects will be shared and some will be “client local”.

Client local means, the content of those objects will not be replicated to the central object store 300, 302. The “screen output” and the “video source” and the “codec” will all be client local objects, so these objects do not “live” in the central object store. Only the “Control object” will be a shared object. The control object contains the controls for “Play”, “Pause”, “Fast Forward”, etc. Furthermore, the control objects contain a “current frame” property, containing the frame that is currently being played. The control object does not contain the whole state of the video application, but only describes the state that is relevant to recreate the state in the event of the client device losing its connection to the application server. The control object contains all the data relevant for describing the state of the video player, this data could be:
- The player state (play, pause, fast forward, etc.),
- The current frame being played,
- The current volume.

The main application loop is essentially the same as in example 1, and will not be discussed here. The reason why only this control information is replicated and not the whole screen data of the application is to save a lot of bandwidth and/or processing power on both the “server” and the “client” part of the infrastructure. The replication of the control object will happen the same way the objects are replicated in the Word Processor application. If the “screen” object would be shared, the raw screen data of the movie would pass with only a minimal amount of compression added, resulting of a
huge data-stream. If a higher compression would be added to the video-stream, it would result in a higher CPU usage at the server part of the infrastructure and it would also introduce additional loss of quality for the video stream.

Example 3: Microwave

In yet another embodiment, the first processing device 2 is arranged to execute a domestic application, such as a microwave operating system.

Most modern microwave oven feature a microprocessor, running some sort of program to cook the food. Such a program can be very straight-forward, like leave the magnetron on for x seconds at y watts, where x and y are chosen by the user. A bit more complex program could be a program we'll call “slow cooking” where the first half of x the magnetron would run at 0.5y Watts and the later half on y Watts. Most microwaves also feature a door, some input controls, a turn table, a ventilator and a light. The traditional microwave can also be looked at in an “Object oriented” way. The following objects could be defined:

- The door; being open or closed
- The control object; representing the input controls
- The magnetron; responsible for heating up the stuff
- The lamp; letting your food shine in the dark
- The ventilator; cooling the unit and expelling odors and smoke produced by the food

The “main application loop” for the simple microwave program looks for example like:

- Read values from control object
- If a start sign is detected and the door is closed:
  - Set the magnetron to the specified amounts of watts.
  - Turn on the lamp
  - Turn on the ventilator
  - Start if the “cook food loop”:
    - Check if somebody pressed “STOP”
    - Check if the specified amount of time has passed
  - Power-down the magnetron
o Turn off the lamp
o Turn of the ventilot

The microwave embodiment can be split up into different implementations. Two of them are discussed below.

First implementation
The first one is the “totally networked microwave” wherein the program logic for the microwave runs on the backend (i.e. the first processing device 2) and the update of the states of the objects will be replicated via the REP protocol, just like it is done in the word processing application discussed above. If the program on the backend enables the magnetron, the REP communication will make sure this update in state will be replicated to the microwave oven. The microwave oven will then turn on the magnetron and start cooking the food, etc. In this way, a user can control her microwave remotely if she has access to the backend. This access may for example be arranged by connecting a second client device to the backend. The second client device may be a computer loaded with for example a control program especially designed to remotely control the microwave. The control program may have screen objects representing control of the actual microwave. The control program will communicate via the REP protocol with the backend. So when the user presses a button on the screen, an object change occurs in the second client device (such as Microwave.Power-on := true); the object change is then replicated to all devices that share the particular object. In this case those devices will be the backend and the first client device. In the first client device 4 (i.e. the microwave), the object change will then cause the operating system 180 of the microwave to start the microwave. The totally networked microwave will only work when a working connection to the application server 2 (i.e. the backend) is present. In this implementation, the objects used by the client device 4 will all be shared objects.

The OS dependent layer 183 of the microwave 4 receives updates of the shared objects. A property of a shared object “heating” could be “power”, this will be called “heating.power”. This property describes the amount of watts that is being emitted by the magnetron (the physical “heating module” of the microwave). The OS dependent layer is arranged to update a certain instruction once the property “heating.power” changes. This instruction would be setting a certain register in the hardware so the magnetron will be set to a certain power level.
Second implementation
In a less-perfect world where the client device 4 is not constantly connected to the backend (i.e. the first processing device 2 and the COS), the invention can still be used to remotely control a microwave application. In this case, the microwave program runs on the backend but it will also run on the microwave 4 itself. This allows the microwave to be a stand-alone device, which can update its state to the backend and central object-store, whenever a connection is available. The difference as compared to the first implementation (i.e. the totally networked microwave) is that in this case, the second processing device 4 (in this case the microwave) is the host of the application, and the backend acts as “client”. Also, all relevant objects will be shared objects, but with a “sync when available” flag set. This flag on an object allows an object to be out-of-sync. However, the application may not use objects which cannot be “certified” as “in-syne”. So the application on the backend can only manipulate the state of the microwave when the objects are “available” and synchronized, so whenever the microwave is disconnected, the state in the central object store cannot change, but the state on the microwave can.

In this way, an implementation of an application is given, that is capable of controlling a microwave remotely, whenever it’s connected to the backend, but is also aware of the fact that it can be disconnected. To avoid overcooking of food when the connection breaks away during a remotely initialized food-cooking process, some safety measures inside the program of the microwave can be implemented.

While specific embodiments of the invention have been described above, it will be appreciated that the invention may be practiced otherwise than as described. For example, the data processing system may comprise a plurality of client devices running the same application, such as a computer game. Each client device will have shared objects which are changed at the different client devices.

These object changes are then sent to the other client using the REP protocol. Furthermore, it is possible to share the state of an operating system instead of an application state.

The descriptions above are intended to be illustrative, not limiting. Thus, it will be apparent to one skilled in the art that modifications may be made to the invention as described without departing from the scope of the claims set out below.
CLAIMS

1. A data processing system (1) comprising:
   - a first processing device (2) comprising a first processor arranged to load a first operating system (40) and arranged to monitor changes of at least a first object, and
   - a second processing device (4) comprising a second processor arranged to load a second operating system (180) and arranged to monitor changes of at least a second object, wherein said first processing device (2) is arranged to send a first object change message to said second processing device (4), when a change of said first object is initiated at said first processing device (2), and wherein said second processing device (4) is arranged to send a second object change message to said first processing device (2), when a change of said second object is initiated at said second processing device (4), and wherein said first and said second processing device (2, 4) are arranged to process changes of said first and second objects so as to copy said first object to said second object after a change of said first object, and so as to copy said second object to said first object after a change of said second object.

2. Data processing system according to claim 1, wherein said first processor is arranged to load a first virtual machine on said first operating system, said first virtual machine being arranged to create said first object change message, and to send said first object change messages to said second processing device.

3. Data processing system according to any of the preceding claims, wherein said second processor is arranged to load a second virtual machine on said second operating system, said second virtual machine being arranged to create said second object change message, and to send said second object change messages to said first processing device.

4. Data processing system according to any of the preceding claims, wherein said first processing device (2) is arranged to execute an application, and said first processor is arranged to translate an assignment made by said application into said first object change message.

5. Data processing system according to claim 4, wherein said first processing device is an
application server arranged to execute a plurality of applications.

6. Data processing system according to claim 4, wherein said first and second object comprise at least one of data and instructions of said application.

7. Data processing system according to any of the preceding claims, wherein said first and said second processing device are arranged to communicate using an internet protocol.

8. Data processing system according to claim 7, wherein said internet protocol is arranged to sent messages via the internet, said messages comprising:
   - object change information,
   - identification information of all devices that share the object involved.

9. Data processing system according to any of the preceding claims, wherein said first and said second processing device are arranged to communicate with a wide area network protocol.

10. Data processing system according to any of the preceding claims, comprising at least a first storage device (302) arranged to communicate with said first processing device (2) and arranged to store said first object.

11. Data processing system according to claim 10, comprising at least a third processing device arranged to, at least in use, execute an application that is identical to said application running on the first processing device, said third processing device being arranged to monitor at least a third object, wherein said first and said third processing device are arranged to send and receive data to and from said storage device (302) so as to copy said first object to said third object and visa versa after a change of said first or said third object respectively.

12. Data processing system according to claim 11, wherein said first processing device comprises an object validator arranged to check if said first object is valid or not, and wherein said first processing device is arranged to load an update for said first object from said storage device only in the situation where said first object is invalid.
13. Data processing system according to claim 10, comprising two or more storage devices arranged to communicate with said first processing device and arranged to store said first object.

14. Data processing system according to any of the preceding claims, wherein said first processing device is arranged to execute a desktop application.

15. Data processing system according to any of the preceding claims, wherein said first processing device is arranged to execute a video application.

16. Data processing system according to any of the preceding claims, wherein said first processing device is arranged to execute a domotics application.

17. Data processing system according to any of the preceding claims, wherein said second processing device (4) is arranged to communicate a format of said second object to the first processing device (2).

18. Use of a data processing system according to any of the preceding claims.

19. Method of processing objects in a data processing network comprising a first processing device (2) and a second processing device (4), said method comprising:
   - monitoring object changes in said first processing device;
   - monitoring object changes in said second processing device;
   - sending object changes initiated at said first processing device to said second processing device;
   - sending object changes initiated at said second processing device to said first processing device.

************
Fig 4
# INTERNATIONAL SEARCH REPORT

## A. CLASSIFICATION OF SUBJECT MATTER

**INV. G06F9/46**

According to International Patent Classification (IPC) or to both national classification and IPC

## B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

**G06F H04L**

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

## Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

**EPO-Internal, WPI Data**

## C. DOCUMENTS CONSIDERED TO BE RELEVANT

<table>
<thead>
<tr>
<th>Category</th>
<th>Citation of document, with indication, where appropriate, of the relevant passages</th>
<th>Relevant to claim No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>EP 0 553 560 A (GPT LIMITED) 4 August 1993 (1993-08-04) the whole document</td>
<td>1-19</td>
</tr>
</tbody>
</table>

F further documents are listed in the continuation of Box C.

See patent family annex.

---

**Date of the actual completion of the international search**

7 July 2006

**Date of mailing of the international search report**

18/07/2006

**Name and mailing address of the ISA/ European Patent Office, P.B. 5015 Patentlaan 2 NL – 2280 HJ Rijswijk**

Tel: (+31-70) 340-2040, Tx: 31 651 epo nl, Fax: (+31-70) 340-0316

**Authorized officer**

Beyer, 0
<table>
<thead>
<tr>
<th>Patent document cited in search report</th>
<th>Publication date</th>
<th>Patent family member(s)</th>
<th>Publication date</th>
</tr>
</thead>
<tbody>
<tr>
<td>US 6195685 B1</td>
<td>27-02-2001</td>
<td>NONE</td>
<td></td>
</tr>
<tr>
<td>EP 0553560 A</td>
<td>04-08-1993</td>
<td>AT 170640 T</td>
<td>15-09-1998</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AU 662784 B2</td>
<td>14-09-1995</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AU 3106193 A</td>
<td>05-08-1993</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CA 2088429 A1</td>
<td>01-08-1993</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CN 1078336 A</td>
<td>10-11-1993</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DE 69226858 D1</td>
<td>08-10-1998</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DE 69226858 T2</td>
<td>21-01-1999</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DK 553560 T3</td>
<td>31-05-1999</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ES 2121829 T3</td>
<td>16-12-1998</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FI 930389 A</td>
<td>01-08-1993</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GB 2263797 A</td>
<td>04-08-1993</td>
</tr>
<tr>
<td></td>
<td></td>
<td>JP 5282165 A</td>
<td>29-10-1993</td>
</tr>
<tr>
<td></td>
<td></td>
<td>US 5734902 A</td>
<td>31-03-1998</td>
</tr>
<tr>
<td>US 6678882 B1</td>
<td>13-01-2004</td>
<td>NONE</td>
<td></td>
</tr>
</tbody>
</table>