The present invention relates to an improved process and product for updating or patching computer files between a plurality of computers. The invention utilizes an error checking algorithm to perform intermediate checks during the updating.
FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

<table>
<thead>
<tr>
<th>AL</th>
<th>Albania</th>
<th>ES</th>
<th>Spain</th>
<th>LS</th>
<th>Lesotho</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM</td>
<td>Armenia</td>
<td>FI</td>
<td>Finland</td>
<td>LT</td>
<td>Lithuania</td>
</tr>
<tr>
<td>AT</td>
<td>Austria</td>
<td>FR</td>
<td>France</td>
<td>LU</td>
<td>Luxembourg</td>
</tr>
<tr>
<td>AU</td>
<td>Australia</td>
<td>GA</td>
<td>Gabon</td>
<td>LV</td>
<td>Latvia</td>
</tr>
<tr>
<td>AZ</td>
<td>Azerbaijan</td>
<td>GB</td>
<td>United Kingdom</td>
<td>MC</td>
<td>Monaco</td>
</tr>
<tr>
<td>BA</td>
<td>Bosnia and Herzegovina</td>
<td>GE</td>
<td>Georgia</td>
<td>MD</td>
<td>Republic of Moldova</td>
</tr>
<tr>
<td>BB</td>
<td>Barbados</td>
<td>GH</td>
<td>Ghana</td>
<td>MG</td>
<td>Madagascar</td>
</tr>
<tr>
<td>BE</td>
<td>Belgium</td>
<td>GN</td>
<td>Guinea</td>
<td>MK</td>
<td>The former Yugoslav</td>
</tr>
<tr>
<td>BF</td>
<td>Burkina Faso</td>
<td>GR</td>
<td>Greece</td>
<td>ML</td>
<td>Mali</td>
</tr>
<tr>
<td>BG</td>
<td>Bulgaria</td>
<td>HR</td>
<td>Croatia</td>
<td>MN</td>
<td>Mongolia</td>
</tr>
<tr>
<td>RJ</td>
<td>Benin</td>
<td>IL</td>
<td>Israel</td>
<td>MR</td>
<td>Mauritania</td>
</tr>
<tr>
<td>BR</td>
<td>Brazil</td>
<td>IS</td>
<td>Iceland</td>
<td>MW</td>
<td>Malawi</td>
</tr>
<tr>
<td>BY</td>
<td>Belarus</td>
<td>IT</td>
<td>Italy</td>
<td>MX</td>
<td>Mexico</td>
</tr>
<tr>
<td>CA</td>
<td>Canada</td>
<td>JP</td>
<td>Japan</td>
<td>NE</td>
<td>Niger</td>
</tr>
<tr>
<td>CF</td>
<td>Central African Republic</td>
<td>KE</td>
<td>Kenya</td>
<td>NL</td>
<td>Netherlands</td>
</tr>
<tr>
<td>CG</td>
<td>Congo</td>
<td>KG</td>
<td>Kyrgyzstan</td>
<td>NO</td>
<td>Norway</td>
</tr>
<tr>
<td>CH</td>
<td>Switzerland</td>
<td>KP</td>
<td>Democratic People's Republic of Korea</td>
<td>NZ</td>
<td>New Zealand</td>
</tr>
<tr>
<td>CI</td>
<td>Côte d'Ivoire</td>
<td>KR</td>
<td>Republic of Korea</td>
<td>PL</td>
<td>Poland</td>
</tr>
<tr>
<td>CM</td>
<td>Cameroon</td>
<td>KZ</td>
<td>Kazakhstan</td>
<td>PT</td>
<td>Portugal</td>
</tr>
<tr>
<td>CN</td>
<td>China</td>
<td>LC</td>
<td>Saint Lucia</td>
<td>RO</td>
<td>Romania</td>
</tr>
<tr>
<td>CU</td>
<td>Cuba</td>
<td>LI</td>
<td>Liechtenstein</td>
<td>RU</td>
<td>Russian Federation</td>
</tr>
<tr>
<td>CZ</td>
<td>Czech Republic</td>
<td>LK</td>
<td>Sri Lanka</td>
<td>SD</td>
<td>Sudan</td>
</tr>
<tr>
<td>DE</td>
<td>Germany</td>
<td>LR</td>
<td>Liberia</td>
<td>SE</td>
<td>Sweden</td>
</tr>
<tr>
<td>DK</td>
<td>Denmark</td>
<td>SI</td>
<td>Slovenia</td>
<td>SG</td>
<td>Singapore</td>
</tr>
<tr>
<td>EE</td>
<td>Estonia</td>
<td>SK</td>
<td>Slovakia</td>
<td>SN</td>
<td>Senegal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SZ</td>
<td>Swaziland</td>
<td>TD</td>
<td>Chad</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TG</td>
<td>Togo</td>
<td>TJ</td>
<td>Tajikistan</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TM</td>
<td>Turkmenistan</td>
<td>TR</td>
<td>Turkey</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TT</td>
<td>Trinidad and Tobago</td>
<td>UA</td>
<td>Ukraine</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UG</td>
<td>Uganda</td>
<td>US</td>
<td>United States of America</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UZ</td>
<td>Uzbekistan</td>
<td>VN</td>
<td>Viet Nam</td>
</tr>
<tr>
<td></td>
<td></td>
<td>YU</td>
<td>Yugoslavia</td>
<td>ZW</td>
<td>Zimbabwe</td>
</tr>
</tbody>
</table>
MICROFICHE APPENDIX REFERENCE

FIELD OF THE INVENTION

The present invention relates to a method for updating large remote system files via an internet connection or other TCP/IP network.

BACKGROUND OF THE INVENTION

Most present computer systems are usually incapable of efficiently updating large computer stored files without the need to replicate the entire file when providing updates or file changes from one data system to another. The initial step of replicating an entire file during an update process is inefficient given that it represents an additional step in an already time consuming procedures. With such an inefficient configuration, the results obtained are generally less than satisfactory because extensive computer resources are required, such as memory, CPU cycles and bandwidth. Even given the more efficient prior art methods, such as calling the "dif" function in the UNIX (a trademark of AT&T) operating system, or alternatively, the running of an entire patch program such as the one marketed under the trademark .RT Patch Professional, owned by Pocket Soft of Houston, TX, U.S.A., no provision is made for an efficient variable size entry to accommodate the inevitably changing sizes of data patch files because, for example, in the case of the "dif" function, one makes a text file which is much larger memory-wise than any patch file, while in .RT Patch Professional, it is believed that one utilizes a much larger executable file which requires more CPU cycles for the creation of files. Often, these and other prior art systems tax already
scarce system resources in terms of requiring excessive CPU cycles and memory, and in the case of a procedure involving the "dif" function, require extensive bandwidth. In order to cope with the present drain on resources when updating files, systems often require various configurations of parallel processors, additional RAM and special bandwidth provisions. Moreover, even with the provision of these expensive resources, certain prior art systems (such as the "dif" function) are less reliable because of the lack of redundancy checks and bandwidth scarcity during transfer can interrupt or otherwise corrupt file transfers between computers during the transfer portion of the updating process, and in the case of .RT Patch Professional, it is believed that the one time use of large, 128 bit error checks rather than the multiple, intermediate redundancy checks of the present invention may result in a greater percentage of mismatches or errors.

The lack of more efficient methods for the patching of files means that, at the very least, updating files is a rather cumbersome or unnecessarily error prone task, given the time consuming, capital intensive logistics involved as well as the chances for incurring error over a period of given time. In order to update file systems, particularly large file systems - whether in the internet or in TCP/IP networks in general - the current methodology essentially requires either: (1) that one transfer or rewrite an entire file with a more recent version; or, at least, (2) that one transmit data transactions and run an update process on the receiving system, locating the records in the change file and committing the updated records without regard for efficiently varying sizes in individual patch file sizes or redundancies in data. The methodology for this typically involves utilizing a binary difference process. Although there are currently several binary difference
file systems in use, the process is frequently the same; one must create an entire copy (i.e., make a mirror image) of the data file for rewriting on the given remote file system in the update process. However, in order for the various prior art systems to accomplish a file update task, at least one of three possible demands are exacted from computer systems: (1) the remote computer must have additional file space (i.e., memory); (2) the remote computer must have processing time (i.e., CPU cycles) available during the update process; and (3) the network itself utilizes a potentially extensive bandwidth during the transfer portion of the update process. Moreover, the above system requirements of the prior art are particularly onerous because the added CPU resources required to perform the whole update process must also be managed by the process itself in order to ensure that the system is not over-utilized during the file updating.

In this regard, there is a real need for a multi-platform compatible process which efficiently modifies the files of systems when one desires the most up-to-date or correct information. Moreover, there is a need to accomplish this task without the necessity of system resources such as extensive bandwidth and memory or a super fast CPU configuration. In particular, however, there is a real need for a method in an internet or other TCP/IP network for efficiently updating large computer stored files, viz., updating large computer stored files without replicating an entire file during file updates in data from one system to another and without requiring burdensome process management which can stem from inefficiencies such as redundant data copying.

Thus there is a need for a cheaper, faster, and more reliable process for effecting file updates that does not require additional computer resources or extensive telecommunications bandwidth to the extent required in the prior art.
SUMMARY OF THE INVENTION

The present invention relates to a controlled binary difference software method of providing small packets of updated data information over the internet or any TCP/IP network to update remote file systems of a programmable apparatus on a continual basis and minimizes the requirements of the remote CPU and memory required to perform the update process as well as the file space required to perform the update process. Accordingly, the present invention provides for a method of optimally controlling the transfer, storage, and retrieval of data in a programmable apparatus such that the most accurate or up-to-date information is readily available.

It is an object of the present invention to provide a method for use in an internet or other TCP/IP network environment for efficiently updating files stored in computer systems of other programmable apparatus(es), without replicating an entire file during file change updates in data from one system to another, and furthermore, without requiring burdensome process management and extensive bandwidth usage. Even more specifically, it is a further object of the present invention to provide a system and method which does not require a remote programmable apparatus to have additional file space (i.e., memory) and processing time (i.e., CPU cycles) available, nor which requires the network itself to use extensive bandwidth even in large file updating operations.

Other objects and advantages will become apparent from the following specification taken in connection with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

SUBSTITUTE SHEET (RULE 26)
FIG. 1' is a diagrammatic representation of a typical TCP/IP configuration between a source computer system and a target computer system, including the CPU and memory portions.

FIG. 1 is a diagrammatic representation of the FilePatchC Usage in C++ language, given a general overview of the Constructor and Destructor Method including a representation of the Create_Patch Method and Apply_Patch Method are integrated in the file update process according to the present invention.

FIG. 2 is a diagrammatic representation of the CreatePatchFile Method of a file update process according to the present invention.

FIG. 3 is a diagrammatic representation of the PatchOut_Create Method including the Source Compare Loop of a file update process according to the present invention.

FIG. 4 is a diagrammatic representation of the PatchOut_Init Method of a file update process according to the present invention.

FIG. 5 is a diagrammatic representation of the PatchC PatchOut_Close Method of a file update process according to the present invention.

FIGS. 6, 6A are a diagrammatic representation of the Source_Read Method of a file update process according to the present invention.

FIG. 7 is a diagrammatic representation of the SourceRXC_Add Data Method of a file update process according to the present invention.

FIG. 8 is a diagrammatic representation of the Source_Compare Method of a file update process according to the present invention.

FIG. 9 is a diagrammatic representation of the Source_FindSync Method of a file update process according to the present invention.
FIG. 10, 10A are a diagrammatic representation of the Source_FindMatch Method including a loop for comparing data rows and a loop for newer data set comparisons.

FIG. 11 is a diagrammatic representation of the Apply_PatchFile Method of a file update process according to the present invention.

FIG. 12 is a diagrammatic representation of the PatchFile_Apply Method of a file update process according to the present invention.

DETAILED DESCRIPTION OF THE PRESENT INVENTION
AND DESCRIPTION OF THE PREFERRED EMBODIMENTS

In general, the present invention provides for a file patching system or method which includes a class of object oriented software which operates in a computer system which comprises a source computer and a remote target computer connected together via a TCP/IP, or internet based configuration. Broadly speaking, the elements of the present invention further include: a means for accessing or reading the data of the Newer Version File (also referred to as the Newer Version and is typically located only in the memory of the reference computer) and the data of the Older Version File (also referred to as the Older Version and is typically located in the memory of the source and target computers); a means for comparing the Older Version File of the reference computer and the Newer Version File of the reference computer in order to create a Patch File having at least one Patch Action; a means for applying a Patch File; and an error checker or error checking means or device. Specifically, the error checker is a means of checking the above-mentioned means for comparing and the applying means to reduce the likelihood of file patching errors.
By way of a preferred embodiment the referenced code relating to the invention which demonstrates only one particular implementation of the present invention, and expressed as a broad overview of the preferred embodiment, the process of the present invention involves, inter alia; (I) the "create portion of the process"; creating a Patch File on a reference computer from the comparison of two corresponding, but different (i.e., in terms of updated status) reference files; and (II) the "apply portion of the process"; applying the Patch File to a corresponding target file (i.e., a file which one desires to update) on a target (remote) computer. Thus, as cross referenced with FIG. 1′, the above survey typically entails the utilization of the aforementioned elements according to the following sequential overview: (1) the means for accessing or reading the data of the Newer Version File (or "reference Newer Version File"; i.e., the version that one wishes to change a given corresponding file to) accesses or reads the data of the reference Newer Version File from the memory means of the reference computer; (2) the means for accessing or reading the data of the Older Version File (or "reference Older Version File"; the reference Older Version is predicated on the assumption that files in the target computer system have not been changed unilaterally, i.e. without the aid of a patch update from the reference computer, as such, the reference Older Version File and the corresponding target Older Version File are deemed identical in content) accesses or reads the data of the reference Older Version File from the memory means of the reference computer; (3) the means for comparing the Older Version File of the reference computer and the Newer Version File of the reference computer compares the Older Version File of the reference computer and Newer Version File of the reference computer to determine correspondence between the two (i.e., that
both files are the same file, but different in terms of update status in as much as one represents the "Newer Version" or updated version of a file, and the other represents the "Older Version" or not-up-to-date version of the same file) as well as to determine the differences between the two files, as this will represent the information to be updated and will be coupled with a Patch Action which together, will comprise the Patch File to be transmitted to the target computer; the means for applying the Patch File including attendant Patch Action(s) to the corresponding target Older Version File located on a target computer memory means, wherein the means includes a means for accurately transmitting during the transfer portion of the file updating process of the Patch File across the connection between the reference computer and the target computer as is known according to those skilled TCP/IP methodology; and (5) utilizing the Error Checker (i.e., an error checking means or device) to check the means of comparing and the applying means to ensure that: (i) the patch process applies to the correct source version (i.e., the correct match between a given reference Newer Version File and the corresponding reference Older Version File) and (ii) the output patched File Version Matches the version used (i.e. the Newer Version) to create the binary difference file.

Although the preferred embodiment provides for a committed reference computer system applying a patch to a given file in a committed target computer system, those skilled in the art will appreciate that it is possible (although generally logistically inefficient) to have a role reversal in the reference/target system dichotomy whereby any unilateral update stemming from a previously termed target computer system could be transmitted as a patch to the previously termed reference computer system with the aid of the present invention. However,
depending on the number of target systems, it may be
difficult to yield consistent results if for example, the
same file were patched by multiple "reference" systems.
Nevertheless, those skilled in the art will further
recognize that the present invention is not limited to a
specific manufacture, but rather, is contemplative of any
and every manufacture causing the computer to perform the
underlying process. Thus, expressed generally, the
present invention encompasses a computer data signal
embodied in a carrier wave (such as electrical wires,
optical fiber cables, air, etc.) comprising the general
objects methods, etc., disclosed herein.

Nevertheless, the present invention
contemplates that a most efficient configuration is for
the reference computer system to have at least four data
components supplied from its memory means (FIG. 1') at
some point in the process: (i) the Older Version File (at
least resident from before commencement of the process
until conclusion of the create portion of the process);
(ii) the Newer Version File (at least during the create
portion of the process); (iii) the Patch File Program or
software itself, i.e., FilePatchC, (at least resident
from before commencement of the process until conclusion
of the create portion of the process); and (iv) the Patch
File itself (at least during the create portion of the
process). Similarly, the target system will have at
least four data components supplied to or from its memory
at some point in the process of the present invention:
(i) the Older Version File (at least resident from before
commencement of the process until conclusion of the
process); (ii) the Patch File program or software itself
(at least resident from before commencement of the
process until conclusion of the process); (iii) the Newer
Version File (at least from after conclusion of the apply
portion of the process); and (iv) the Patch File itself
(at least during the apply portion of the process until conclusion of the process).

The present invention also contemplates that, any point above, the supplying or transmitting of data may be encrypted as is commonly known in the art. In the preferred embodiment, the reference system will use a commercially available compression program utilizing any encryption standard, such as DES or RSA, which can be used to compress and encrypt the Patch File to be transmitted to the target computer system. Alternatively, the present invention also provides for the use of compression and encryption as described in U.S. Patent No. 5,022,076, the specification of which is incorporated herein by reference.

In a preferred embodiment, the file update or "Patch" of the present invention will operate from a means for supplying digital information such as a computer memory encoded with executable instructions representing the program or software that will cause the computer to update files either internally or remotely. Updating of files of a remote or target computer system from a reference computer system may be accomplished through any conventional means known to those skilled in the art, but in a preferred embodiment, the remote or target system will be connected to the base or reference computer system via a modem set-up which is connected to a TCP/IP network such as the internet. More specific details of the particular configurations are detailed in the specifications of patents such as U.S. Patent Nos. 5,163,131 and 5,729,735, the specifications of which are hereby incorporated by reference.

Although the means for supplying digital information for such a computer-related system may comprise diverse alternatives such as information storage devices (i.e., local sources such as a magnetic disk drive, magnetic tape, semiconductor memory, holographic
embodiments, etc.), or transmission lines carrying live data from a remote source, in a preferred embodiment the means for supplying digital information will comprise a computer system of at least average (as currently measured) means, such as one which has a CPU marketed under the Intel Corporation trademark Pentium and having a clock speed of say, at least 133 Mhz. and having approximately 16 Megabytes RAM and 2.0 Gigabytes of hard drive space, an exemplary structure of which is disclosed in U.S. Patent No. 5,404,559, the disclosure of which is incorporated herein. In an alternative embodiment, the means for supplying digital information will comprise a RAID type arrangement of computer memory, such as that described in U.S. Pat No. 5,101,492, the specification of which is incorporated herein by reference. Nevertheless, the minimum required embodiment is at the very least, the equivalent of a 32 bit computer system having a 486-type Intel 33 Mhz microprocessor having little or no RAM and a 20 Megabyte hard drive.

In addition, the present invention contemplates multi-platform compatibility between various programmable apparatus(es). Specifically, the invention may be used in all types of computerized systems, ranging from stand-alone internet based computers to LAN/WAN/MAN network computers, regardless of the operating system or hardware used or size of file(s) needed to be updated. By way of example, the present object class, termed FilePatchC, is compatible with Windows, UNIX and Unicode, and uses the standard C library for File I/O, Memory Management, and Platform Independence. The source code explicitly supports Microsoft Visual C++ compiler for Windows 95/NT and GNU Cpp Compiler for Linux/UNIX.

The present invention contemplates various embodiments, and as such, may be utilized in several different ways.
For example, the present invention may be viewed as a patching system for patching files in connected computer systems where the patching system has at least a means for reading a Newer Version File in a reference computer and a corresponding Older Version File in a reference computer, as well as a comparison means for determining which of the corresponding Older Version Files in a computer system need to be patched, a means for creating or writing a Patch File containing at least one Patch Action based on the results of the above comparison (whereby the creating means may be included within the comparison means above), and means for applying the Patch File to an Older Version File in a target computer system to create a Patched Version File. Such a system would ideally include an error checker for checking for any differences between the reference Newer Version File and the reference Older Version File, and for checking any differences between the Patched Version File and the Newer Version File.

Similarly, the structure of the present invention might constitute a patching system for patching files in computer systems which is coupled to a control computer via a TCP/IP connection. At least one, if not both computers will have a memory portion, for storing or using software for reading a reference Newer Version File and a reference Older Version File, as well as software for comparing the reference Newer Version File with the reference Older Version File in order to prepare a Patch File containing at least one Patch Action, and software for applying the Patch File to a target Older Version File in order to create a Patched Version File and software for error checking the above steps in the patching system.

Alternatively, the present invention may be viewed as method of patching files in remote computer systems. Such a method would preferably comprise the
steps of accessing a reference Newer Version File in a
reference computer system, accessing a corresponding
reference Older Version File in the reference computer
system, error checking the correspondence of the
reference Older Version File with the reference Newer
Version File, comparing the reference Older Version File
with the reference Newer Version File in order to prepare
a Patch File having at least one patch action, applying
the Patch File to a target Older Version to create a
Patched Version File, and error checking the Patched
Version with the Newer Version.

Expressed as a file patcher for patching files
in connected computer systems, the present invention has
means for reading a reference Newer Version File on a
memory location of a reference computer, as well as a
means for reading a corresponding reference Older Version
File in the memory location of the reference computer,
and means for comparing the reference Older Version File
with the reference Newer Version File for purposes of
preparing a Patch File having at least one Patch Action,
a means of applying the Patch File to a corresponding
Older Version in the memory of a target computer, as well
as a means of error checking the comparing means and the
applying means so as to reduce the likelihood of
incurring patch errors.

When expressed as a file patcher the file
patcher may be stored via storage media. The storage
media would comprise a first plurality of binary values
for reading a reference Newer Version File on a reference
computer, a second plurality of binary values for reading
a corresponding reference Older Version File on a
reference computer, a third plurality of binary values
for comparing the reference Newer Version File with
corresponding reference Older Version File so as to
Create a Patch File having at least one Patch Action, as
well as a fourth plurality of binary values for applying
said Patch File to a corresponding Older Version File on a target computer to create a Patched Version File, and a fifth plurality of binary values for preventing errors by the file patcher.

By contrast, the present invention may also be embodied as a method of manufacturing data management systems. Ideally this method would comprise the steps of reading a reference Newer Version File of a reference computer system, reading a corresponding reference Older Version File of a reference computer, comparing the reference Older Version File with the reference Newer Version File in order to prepare a Patch File having at least one Patch Action, applying the Patch File to the corresponding target Older Version File of a target computer to create a Patched Version File, and also including error checking the above method to minimize the likelihood of errors.

Additionally, the present invention may be viewed as an object oriented data patching data structure stored in a memory for access when patching files in a remote computer system having the following steps of accessing a method on a reference computer, said accessing method having a read method stored therein for accessing a reference Newer Version File and corresponding reference Older Version File, as well as a comparing method for comparing the reference Newer Version File and corresponding reference Older Version File for creating a Patch File data structure having at least one set of data representing at least one Patch Action, and an application method for applying the Patch File data to a corresponding target Older Version File in order to create a Patched Version File, an error checking method for minimizing the likelihood of errors in implementing the comparing and application methods.

The object class FilePatchC of the present invention provides for a plurality of binary values which
reside in and operate on computer systems, the values of which are collectively represented and classified as discrete objects, each of which manipulates or operates a certain data structure. In terms of the specific operation of the present invention, FilePatchC creates (or applies) a binary difference file by comparing the hexadecimal bytes of two files. The electronic Error Checker, also termed a difference file (for preventing errors by the file patcher), error checking, an error checker, an error checking method, or means for checking errors, utilizes 32 bit Rotating eXclusive-Or Checks (RXC), the algorithm of which is referenced infra, to first, ensure that the patch only applies to the correct source version, and later, as well as to ensure that the output patched file matches the version used to create the binary difference file. FilePatchC also defines a parameter to allow the creation of intermediate RXCs in order to ensure a greater chance of a version match.

More specifically, FilePatchC takes a series of parameters which can optimize the matching process. In optimizing this matching process through a series of parameters, the file (error) checker used in FilePatchC repeatedly takes the 32 bit RXC algorithm at a certain number of times, from beginning to end, after each independent patch action occurs. This intermediate file (error) check is done to ensure that the version which has been identified as the correct file and which is currently being worked on is, in fact, the correct file. Later on, this same file checker is used to ensure that the final Patched Version File matches the Newer Version File. FilePatchC will take a series of parameters for each patch action, and thus, the algorithm used will take 32 bit RXCs for each patch action taken. Thus, generally speaking, the error checking means checks the accessing (i.e., the comparing (and/or creating) step of a Patch File) of the corresponding reference files in the
reference computer system as well as the applying of the Patch File to the target computer system.

Thus, with regard to the specific operation of the RXC algorithm and the match finder methodology of the present invention, the following shows a description of the RXC algorithm and the FindMatch algorithm.

The RXC algorithm, called as SourceRxc_AddData and having the C++ variables (ByT num, DatT data), adds a new data value <data> to the 32 bit RXC Value for File Version <num> and may assume a particular data structure for maximization of the algorithm listed below. During the Patch Apply Process, Intermediate RXC Values are checked for failure to match according to the following equation:

\[ RXC = ( ( RXC \text{ ShiftRight} \ 8 \text{ bits} ) \ OR \ ( RXC \text{ ShiftLeft} \ 24 \text{ bits} ) ) \xor \text{ Data}. \]

This RXC equation may be illustrated by the following example where we imagine a data stream A B C D E F G H I which we will want to match against later, and assuming a long of 4 bytes, we can follow the progression of the initial data being manipulated by the 32 bit RXC algorithm.

Thus, a 32 Bit RXC builds as follows (where the symbol " \( ^{\wedge} \) " signifies the XOR operand:

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>Low</td>
<td>Result</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Init:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>. . .</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

A: 0 0 0 0 \( ^{\wedge} \) A = 0 0 0 0 A (A)

B: A 0 0 0 \( ^{\wedge} \) B = A 0 0 0 B (B)
C : B A 0 0 \wedge C = B A 0 C \quad (C)
D : C B A 0 \wedge D = C B A D \quad (D)
E : D C B A \wedge E = D C B A \wedge E \quad (E)
F : A \wedge E D C B \wedge F = A \wedge E D C B \wedge F \quad (F)
G : B \wedge F A \wedge E D C \wedge G = B \wedge F A \wedge E D C \wedge G \quad (G)
H : C \wedge G B \wedge F A \wedge E D \wedge H = C \wedge G B \wedge F A \wedge E D \wedge H \quad (H)
I : D \wedge H C \wedge G B \wedge F A \wedge E \wedge I = D \wedge H C \wedge G B \wedge F A \wedge E \wedge I \quad (I)

By contrast, current prior art methodology does not use the intermediate file match, but instead uses a one-time 128 bit file match at the end of an entire read process. This prior art methodology is thought to have at least two inherent weaknesses;

(1) use of a 128 bit CRC or Check Sum file process does not utilize a standard C library for File I/O (i.e., a 32 bit size); and

(2) the use of a 128 bit CRC or Check Sum file process does not maximize assurances matching of files because it only samples the identified file(s) once. (i.e., at the end of the process.)

As for the match finder means or methodology, herein also termed the FindMatch Method, the source code calls this particular method as Flgt Source_FindMatch, the purpose of which is to initialize the attendant values (DwdT startPos = 0, DwdT bufSize = 0, WrdT match = 0). The mission of the Find Match as part of the Comparison Process (i.e., the Source_Compare Method), is to compare the data of the Older Version File with the
data of the Newer Version File when preparing a patch file for a Patch Action (i.e., the Patch_Apply Method).

To commence process of the present invention, in a preferred embodiment one will use an operating system call to call up the FilePatchC program, but alternatively, one may call the FilePatchC program from another program with an execute command or from a batch file process by invoking a command such as "binpatch_OlderVersionFile_NewerVersionFile_patchfile/M".

In any case, when invoking the FilePatchC program at the beginning, the FilePatchC program and any concurrent program or necessary operating system will reside in the memory portion of the reference computer (FIG. 1''), preferably in a hard drive, if any exist.

In order to prepare the patch file, the routine as further detailed in the example below, compares the data of the Older Version File with data of the Newer Version File and identifies the differences between the two files (if any) and determines the particular information that needs to be changed in order to effect the updating of the Older Version File. The process begins by searching the Data Buffers, starting at the start position (startPos), and continues until the buffer size (bufSize) is reached. The process continues by looking for a match according to the default match size, or according to a customized match size (if passed).

The process then calls a buffer values search to compare the buffer positions. When a match is found, the match determines the full size from the match to end of the buffer. Having found a match, the PatchOut_Action is called to Output a Patch Action and the Current Status is returned.

The following is an example of how the Source_FindMatch Method operates, as referenced according to the Key, Definitions, and Parameters set forth below:
--- Key
BP = Buffer Positions, FP = File Positions, (A) = Version A, (B) = Version B

Assuming two data streams;

--- the definitions of the assumed data streams, Version A and Version B are;

FP: (A)    FP: (B)
01:A 01:A
02:B 02:B
03:C 03:C
04:D 04:Z
05:E 05:Y
06:F 06:X
07:G 07:W
08:H 08:G
09:I 09:H
10:J 10:I
11:K 11:J
12:L 12:K
13:M 13:L
14:N 14:D
15:O 15:E
16:P 16:F
17:Q 17:G
18:R 18:P
19:S 19:Q
20:T 20:R
21:U 21:S
22:V 22:T
23:W 23:U
24:X 24:V
25:Y 25:W
26:Z 26:X
27:Y
28:K
29:Z

--- Wherein the Basic Patch Parameters are:


In order to create a Patch, the following details matching process against the above data streams:

Write Patch Action : Header
Load First Buffer Set

<table>
<thead>
<tr>
<th>10</th>
<th>BP</th>
<th>FP: (A)</th>
<th>FP: (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01:A</td>
<td>01:A</td>
<td>Match</td>
</tr>
<tr>
<td>01</td>
<td>02:B</td>
<td>02:B</td>
<td>Match</td>
</tr>
<tr>
<td>02</td>
<td>03:C</td>
<td>03:C</td>
<td>Match</td>
</tr>
<tr>
<td>03</td>
<td>04:D</td>
<td>04:Z</td>
<td>No Match -&gt; Reload Buffer, Keep A(03) and Keep B(03)</td>
</tr>
<tr>
<td>04</td>
<td>05:E</td>
<td>05:Y</td>
<td></td>
</tr>
<tr>
<td>05</td>
<td>06:F</td>
<td>06:X</td>
<td></td>
</tr>
<tr>
<td>06</td>
<td>07:G</td>
<td>07:W</td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>08:H</td>
<td>08:G</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>09:I</td>
<td>09:H</td>
<td></td>
</tr>
<tr>
<td>09</td>
<td>10:J</td>
<td>10:I</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11:K</td>
<td>11:J</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>12:L</td>
<td>12:K</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13:M</td>
<td>13:L</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>13:N</td>
<td>14:D</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>15:O</td>
<td>15:E</td>
<td></td>
</tr>
</tbody>
</table>

---- ---- Match Check Overflow Area

| 15 | 16:Q | 16:F   |
| 16 | 17:Q | 17:G   |
| 30 | 17:R | 18:P   |
| 18 | 19:S | 19:Q   |
--- Reload Buffer Set, Keeping Previous A(03), B(03)

BP    FP:(A)    FP:(B)
00 04:D 04:Z  Compare A(00)..A(14) to B(00) : No Match
      Compare B(00)..B(14) to A(00) : False Match at B(10) Fails at B(14) : No Match
01 05:E 05:Y  Compare A(00)..A(14) to B(01) : No Match
      Compare B(00)..B(14) to A(01) : False Match at B(11) Fails at B(14) : No Match
02 06:F 06:X  Compare A(00)..A(14) to B(02) : No Match
      Compare B(00)..B(14) to A(02) : False Match at B(12) Fails at B(14) : No Match
03 07:G 07:W  Compare A(00)..A(14) to B(03) : No Match
      Compare B(00)..B(14) to A(03) : Match 6 at B(04)..B(09)
Write Patch Action : At A:04, Insert "ZYXW", Skip to A:07
      Reload Buffer, Keep A(09) and Keep B(10)

04 08:H 08:G
05 09:I 09:H
06 10:J 10:I
07 11:K 11:J
30 08:12:L 12:K
09 13:M 13:L
10 14:N 14:D
11 15:O 15:E
12 16:P 16:F
Match Check Overflow Area

--- Reload Buffer Set, Keeping Previous A(09), B(10)

BP FP:(A) FP:(B)
10 00 13:M 14:D Compare A(00)..A(13) to B(00) : No Match
     Compare B(00)..B(14) to A(00) : No Match
01 14:N 15:E Compare A(00)..A(13) to B(01) : No Match
     Compare B(00)..B(14) to A(01) : No Match
15
20 15:O 16:F Compare A(00)..A(13) to B(02) : No Match
     Compare B(00)..B(14) to A(02) : No Match
03 16:P 17:G Compare A(00)..A(13) to B(03) : No Match
     Compare B(00)..B(14) to A(03) : Match 10 at B(04)..B(13)
     Write Patch Action : At A:13, Insert "DEFG", Skip to A:16
     Reload Buffer, Keep A(13) and Keep B(14)
25
30 04 17:Q 18:P
05 18:R 19:Q
06 19:S 20:R
07 20:T 21:S
08 21:U 22:T
09 22:V 23:U
10 23:W 24:V
11 24:X 25:W
12 25:Y 26:X
13 26:Z 27:Y
14 28:K
-- ---- ---- Match Check Overflow Area
15 29:Z

--- Reload Buffer Set, Keeping Previous A(13), B(14)

00 26:Z 28:K Compare A(00) .. A(00) to B(00) : No Match

15 Compare B(00) .. B(01) to A(00) : Match

1 (and End of File) at B(01) .. B(01)

Write Patch Action : At A:26, Insert "K"

01 29:Z
-- ---- ---- End of Buffers (End of Files)

20 --- Reload Buffer Returns End of File
Write Patch Action : Footer

Currently, this process has redundant match checks because of the possibility of increasing the buffer size in cases where the process is not finding a match in the buffers. However, all operations in the above example will eventually find a match.

Another embodiment, involving an enhanced pyramidal search scheme, will first check the data already in the buffers with the new data in buffers, using a similar check process.
The basic check of the New Data for matches will increase the first position of the comparison sets by 1 for each check. As below:

BP FP:(A) FP:(B)

5 00 13:M 14:D Compare A(00)..A(13) to B(00) : No Match
     Compare B(01)..B(14) to A(00) : No Match

01 14:N 15:E Compare A(01)..A(13) to B(01) : No Match
     Compare B(02)..B(14) to A(01) : No Match

02 15:O 16:F Compare A(02)..A(13) to B(02) : No Match
     Compare B(03)..B(14) to A(02) : No Match

03 16:P 17:G Compare A(03)..A(13) to B(03) : No Match
     Compare B(04)..B(14) to A(03) : Match
     10 at B(04)..B(13)

Write Patch Action: At A:13, Insert "DEFG", Skip to A:16
Reload Buffer, Keep A(13) and Keep B(14)
FilePatchC has default parameters already set to optimal values for patching most programs and text files for Microsoft® and Unix®/Linux® operating systems because it utilizes a standard C library for File I/O, memory management and platform independence. In addition, the use of the intermediate 32 bit rotating Xor Checks (RXC) also ensures a greater chance of file identification matches.

More specifically, these particular goals are accomplished because the parameters that FilePatch C defines control the sizes of the computer memory buffers utilized by the file checking device algorithm (See FIGS. 6,6A). For example, the amount of data that a patch file covers is often less than a standard word boundary (i.e., in the case of an unsigned short integer of two bytes, a value only up to 65,535 to is held, and in the case of an unsigned long integer of 4 bytes a value up to 4,294,967,295 is held). Assuming the common boundary of an unsigned short integer, the amount of data that a Patch File will cover is often less than 65,535, so present invention provides for a variable size entry to represent the varying amount of data that exists in a Patch File. In contrast to prior art methodologies which always utilize an entire word up to the standard boundary, the present invention accommodates smaller size patch files by using less than a full word. The use of less than a full word means that the present invention utilizes an efficient patch action allocation scheme.
which provides for additional patch actions to be inserted in the bit field to fill up the "saved" space. Adding more patch actions for inclusion in a bit field is believed to be an inherently more efficient procedure which not only saves space, but also CPU cycles. The particulars regarding the executing of a type of instructions which the present invention would manipulate in producing the file update are well known in the art, for example, as detailed in U.S. Patent No. 4,775,978, the specification of which is hereby incorporated by reference.

Thus, given all of the aforementioned, the present invention may utilize, say, a 32 bit entry on a given Patch File, but if the Patch File Action is not a long word then the patch entry will change to 16 bits. This overall change from 4 bytes to 2 bytes makes a slightly smaller Patch File. Furthermore, the method of providing variable size entries for a Patch File representation is accomplished by using bits, rather than bytes, to represent the action to be taken in a patch field (i.e., the area in a Older Version File to be patched via the Patch Action into a properly patched area of an expected Patched File.) For example in the present invention; 1 bit will indicate a footer; 1 bit will represent a skip action; 1 bit may indicate the inclusion of an intermediate Xor value (only if necessary i.e., if a Patch Action is required) and 1 bit will represent whether to use a long or regular word. Moreover, all 8 bits are not necessarily assigned, something which provides for the ability to add more functionality by the inclusion of more patch actions, thereby saving even more space and speeding up processing activity. This is believed to be in contrast to the prior art (such as .RT Patch Professional), where each and every entry is always the same size; i.e., 1 byte is always used to indicate the Patch Action to be taken, regardless of whether an
action is actually needed; 4 bytes are used to indicate the length of the patch file, 16 bytes are used for the CRC check, etc, or in the case of a "dif" function, each entry, although differing in size, contains large text descriptions. In sum, the present invention provides smaller entries for smaller actions, and eliminates redundancies in data copying by not requiring or creating a patch entry for every patch action because a copy is automatically assumed for the appropriate sections instead of actually explicitly creating a copy action.

Also, the aforementioned goals are accomplished because the minimal amount of data that constitutes a probable match is controlled by the particular parameters that FilePatchC defines in the match finder of the comparison means. The amount of data that constitutes a probable match is an important part of the preferred embodiment because subject programs and Text Files often have small sections of redundant data which could incorrectly appear as a match. Thus, requiring matching sections to have a greater length avoids mistakenly identifying a "match" in cases of redundant data.

As detailed in FIGS. 9,10 & 10A the match search of the present invention, as configured, proceeds until it finds a match in the current buffers. When the match search cannot find a match in the current buffers, it automatically increases the buffer size of the search area and loads additional amounts of file information into memory.

The match search process of the present invention uses a pyramidal search algorithm or scheme which ensures that the Comparison Method or process (see FIGS. 8,9 & 10) checks a section of data no more than once. Furthermore, when the buffer size increases, the match search also ensures that the Comparison process considers all combinations of new data, all without any redundancy.
Next, when the match search process finds a "match" it assigns a new match point to that particular block of data. When the match search assigns the new match point it outputs a data entry to a patch file (a "Patch Action") which contains a one-bit piece of code representing the type of information difference. For example, the code representing the information difference for an unneeded or outdated file section would have a code tag denominated as "Delete This Section Range", alternatively, where a block of data is needed as part of the new, updated file to be applied, the code tag would be "Insert the Following Data". The process of the present invention also provides for a code tag denominated such as "Skip Data" or "Skip Older Data" so as to avoid making a mirror-image replication of an entire file as often required in prior art methodology. This provision saves CPU cycles and requires less memory and bandwidth.

Moreover, the above data coding also contains a buffer size flag so as to specify whether the length of a given position is to use a word or double word. Such a provision allows the difference file to use smaller sized entries for smaller changes.

Regarding a more specific overview of the FilePatchC program, the FilePatchC Usage is a preliminary reference point. FilePatchC Usage is determined by BinPatch.cpp, the code of which details examples of how to use the classes developed within the FilePatchC program and FilePatchC.h. FilePatchC is generally concerned with beginning and end declarations: (I) the FilePatchC Constructor Method declared at the beginning; and (II) the FilePatchC Destructor Method declared at the end.

FilePatchC.h and FilePatchC.cpp both determine the operations of both of these broad steps. FilePatchC.h defines the basic class definition including definitions
of methods, Status values, parameter values, and additional values. FilePatch.cpp defines what the source code of the class does by implementing the source code itself.

Regarding the overall usage of process of FilePatch C, 4 main steps are involved:
(1) declaring the Constructor parameter;
(2) creating a Patch File;
(3) applying a Patch File; and
(4) implementing the Destructor. Each of these steps will be examined in detail below, but may be generally related to each other as indicated in FIG. 1.

FIG. 1, With further reference to the present invention begins by creating an instance of FilePatchC with the Constructor Method. As detailed previously, the Constructor Method represents a preliminary allocation, the counter-point of which is the destructor method which is implemented at the end of the FilePatchC process. The constructor method begins by allocating the memory in a buffer of a computer for the class structure and virtual method pointers.

After the proper memory has been allocated, the initial FilePatchC Parameters are assigned and normalized. With continuing reference to FIG. 1, the following parameters are processed in the above task as follows:

(a) a buffer size comparison parameter is assigned and normalized for the comparison of buffer sizes;

(b) a "hit or miss" buffer parameter is assigned and normalized as a provision for when the maximum buffers are loaded when searching for the next match;

(c) a RXC Spacing parameter is assigned and normalized as preparation for an Output Xor Check which
is to be run when a certain number of bytes are processed in a given beginning, intermediate or ultimate step;

(d) a match minimum parameter is assigned and normalized for a minimum number of bytes found when determining a match;

(e) a small buffer parameter is assigned and normalized as a provision for the size of the data needed to check for small differences between versions; and

(f) a small minimum parameter is assigned and normalized as preparation for the minimum number of bytes needed to match for a small difference.

Once the Constructor Method has been implemented, the Constructor is exited and the process moves on to CreatePatchFile, wherein the CreatePatchFile Method creates from zero to any number of Patch Files, depending on the number of files to be updated in a given target system. In doing so, the CreatePatchFile Method utilizes the following discrete Methods or Sub-Methods:

(1) the PatchOut_Create Method; (2) the PatchOut_Init Method; (3) the PatchOut_Close Method; (4) the Source Read Method; (5) the SourceRXC_Add Data Method; (6) the Source_Compare Method; (7) the Source_FindSync Method; and (8) the Source_Find Match Method.

As can be readily ascertained from FIG. 2, the above Sub-Methods (1) through (8) are all functions of the CreatePatchFile Method itself. Specifically, as seen in FIG 2., once the file pointers are established for all files encountered during execution, the CreatePatchFile Method is called.

Turning to FIG.4, once the PatchOut_Create Method is called, the PatchOut_Init is immediately called as a preliminary step. As seen in FIG.4, the PatchOut_Init sets the internal values of the computer hardware for the Patch File Header, and then notifies the CreateProcessStatus Virtual Method, after which the Patch File Header is written. The purpose of the CreateProcess
Status Virtual Method is for BinPatch.c to specify how the status is displayed to the user.

Once this write has been successful, the PatchOut_Init Method is exited, the process proceeds to initialize the pointers of the buffer in the computer to NULL, while the buffer is set to zero (0) which is by default the first position. Immediately thereafter, the Xor Check values are initialized for both the Older and Newer versions. For reference purposes, the terms "Older Version" (or "Older Version File") and "Newer Version" (or "Newer Version File"), as used throughout, shall mean the file version to be most up to date file version, respectively. The process of the present invention uses the Newer Version to compare the Older Version in order to form a modified difference file called a Patch File which contains Patch Actions which specify how to apply the differences in the Patch File to the Older Version in order to create a Patched Version File. The Patched Version File, which is the end result of the process of the present invention, should be ultimately identical to the Newer Version File.

Next, a loop is entered for the Source_Compare Method. The Source_Compare loop comprises the following 3 sub-steps: (i) calling the Source_Read Method for the Older Version; (ii) calling the Source_Read Method for the Newer Version; and (iii) calling the Source_Compare Method.

The first sub-step, calling the Source_Read Method for the Older Version is a means for accessing data from both the Newer and Older Version Files. Depending on the particular point in the process, whether dealing with the Newer Version or Older Version File, it entails reading data from the file while tracking the Xor Check and allocating the proper buffer memory in the computer. More specifically, as seen in FIGS. 6-6A, the Source_Read Method begins by pulling the Source_Read
parameters from the stack set in the buffer of the computer. In particular, this means that a Read function is executed on the Older and New file buffers and additional buffers are created on the current Read with a default setting of zero (0).

Thus, if no file buffer exists the additional buffers are reset to zero (0), the number of existing buffers is set to one (1), the buffer size parameters are determined, and the buffer memory is allocated and assigned a pointer. Thereafter, if no file buffer exists, the status will be read as a failure and the Source_Read Method will be exited; but if a file buffer does in fact exist at this point, the Buffer data values will be reset to empty (i.e., zero (0)) and will proceed to the Additional Buffers determination step.

Alternatively, however, if after pulling the Source_Read parameters from the stack a file buffer exists, the process will proceed instead directly to the additional buffers determination.

Turning to the aforementioned Additional Buffers determination step, should there be any additional buffers to be added, the process will proceed to implement an Additional Buffer Operation. In the event, however, that there are no additional buffers to be added the process will carry through with a No Additional Buffer Operation.

In implementing the No Buffer Operation, an initial determination is made as to whether the Buffer Data Keep is greater than zero (0), but less than that of the actual Buffer length. Should the Buffer Data Keep be greater than zero, but less than the actual Buffer Boundary Length, the data at the Keep Position will be subtracted from the Buffer Length, then the data at Keep Position will next be moved to the beginning of the buffer, and after that, the current file position is incremented to the Keep Position. However, should the
Buffer Data keep not be less than the Buffer Boundary Length, then the process proceeds to increment the current file position to the end of the Buffer Boundary Length, and finally resets the buffer to empty, i.e., zero (0).

Either way, the No Additional Buffer operation of the Source_Read Method next proceeds to reset the Buffer Data in the Keep to the First Position, i.e., zero (0). After this, a determination is made as to whether there is more than one buffer.

In the case of non-multiple buffers, the No Additional Buffer Operation directly terminates, but in the case of multiple buffers, the necessary buffer amount is gleaned from the current data in the buffer. If the necessary number of buffers is less than the current number of buffers, the buffer size is reset, but the necessary number of buffers is not less than the current number, to the necessary size and the buffer memory is reallocated and the buffer pointer is reset if not the No Additional Buffer Operation simply ends. However, once the buffer size is reset to the necessary size, the buffer memory is reallocated and the Buffer Pointer reset, a determination is made as to whether a file buffer must exist in order to complete the No Additional Buffers operation. If a file buffer does not exist, the Status will be set to Failure and the Source_Read Method will be exited. Alternatively however, if the file buffer exists, the "No Additional Buffer" operation simply ends.

By contrast, if in the aforementioned additional buffer determination it is instead determined that additional buffers need to be added, then the Additional Buffer Operation is implemented, beginning with the following steps: (i) incrementing the count of buffers; (ii) normalizing the count to the maximum buffers allowed; (iii) determining the buffer size
parameters; and (iv) reallocating the buffer memory and reassigning the buffer pointer.

Having completed steps (i) - (iv) above, the Additional Buffer Determination next checks whether a file buffer exists. If none exist at this point, then the status is set to Failure and the Source_Read Method exited. However, if a buffer does exist, then the Buffer Data keep position is reset to first position (i.e. zero (0)) before the "Additional Buffer" branch is terminated.

Irrespective of whether a given execution of the process has implemented the No Additional Buffer operation or the Additional Buffer operation, the next step is to ascertain that the proper file buffer provisions have already been implemented so as to ensure that the Data Length is less than the presently allocated Buffer Size. If the Data Length is not less than the Buffer Size, then the Source_Read Method is exited and the Status is returned.

However, if the Data Length is smaller than the Buffer Size, then the appropriate data block is read to fill the empty area of the File Buffer so as to make the most efficient use of all available space. In addition, the CreateProcessStatus Virtual Method is notified. The purpose of the CreateProcessStatus Virtual Method is to allow BinPatch.c to specify how the status is displayed to the user.

Next, the Read function is checked to see if anything less than the full buffer is being read. If less than the full buffer is being read, a check is then made to see if an I/O error has occurred, at which point if such an error has actually occurred, the status is set to Failure and the Method Read_Source is exited.

Assuming no such I/O error has occurred, the process continues and a loop is begun through the New Data. In general, the New Data is defined as the data from the New Version.
Similarly, if the check of the Read function determines that the full buffer is being read, the loop is begun through the New Data. The purpose of this New Data loop is to cycle through the new data while implementing the RXC algorithm in order to minimize the likelihood of incurring any patch errors.

First, the New Data loop begins by calling the SourceRXC_Add Data Method to add values to the Xor Check. With specific reference to FIG. 7, the SourceRXC_AddData Method implements the RXC algorithm beginning by pulling the buffer number and data from the stack. Once this is done, the RXC algorithm is defined by the aforementioned example. Having done this, the current file position counter is incremented. Next, the RXC is checked for a match, and will either reset the Status to Failure in the event of no match, or will directly reset the File Check Position before exiting the SourceRXC_Add Data Method.

Once the SourceRXC_Add Data Method has been exited, a check is made to ascertain as to whether the end of the New Data has been reached. If the end of the New Data has not been reached, the Loop through New Data begins again, but if the end of the New Data has in fact been reached, the New Data Loop terminates and the Source Read Method is exited and the Status returned.

The next step after exiting the Source_Read Method for the Older Version is to assume the identical Source_Read Method progression as detailed above, and implement it for the Newer Version. After the Source Read Method for the Newer Version has been called and fully implemented, the next stage is to call the Source Compare Method.

As Seen in FIG. 8, the Source_Compare Method, begins by determining whether the current Status is set to failure, in which case the Source_Compare Method would be exited. Once it has been ascertained that the current
status is not set to failure, the Comparison Data Size is set to the Smallest Buffer Size.

After the Comparison Data size is set to the Smallest Buffer Size, a series of three (3) possible
determinations may be made as to whether the following quantities exist: (a) an Index; (b) a Buffer Length for
the older version greater than zero; and (c) a Buffer Length for the Newer Version greater than zero.

In the first step for all cases, the buffer will be examined to determine the existence of data.

If data exists in the buffer (i.e., Data Size > 0), the Index is set to the position of the first
non-matching byte in the Buffers or Buffer Size, then the buffer is checked for an index. If there is an index,
then the Internal Values are set to skip the existing matches (as indicated by the present values in the index)
during the next Source_Read. It is believed that setting the Internal Values to skip the existing matches
eliminates the need for the prior art "mirror-image" copying and reduces the system requirements for memory
and CPU cycles because a patch entry is not created for each entry; unnecessary entries are eliminated by
assuming that the copying of relevant portions is automatically effected without requiring an explicit
patch entry each time. After setting the Internal Values, the Status is set to Success and the Source
Compare Method is exited. However, if data does not exist in the Index, then the Source_FindSync Method
(FIG. 9) is called. The purpose of the Source_FindSync Method is to search enough data so as to permit the
proper comparison of two files after one has encountered a difference between the two files (i.e., between the
Older and Newer Versions). In the present case, properly comparing files entails selecting large enough blocks of
data for each round of comparison so as not to permit the differentiation process in the Source_Compare Method to
be compromised or otherwise confused by the blocks of redundant data which may be present in files. In a preferred embodiment, the default setting of blocks of data that need to be selected are approximately double the typical patch file entry. Alternatively, the user may reconfigure the default setting so as to alter the minimum size block of data used in the above process.

The selection of these minimum blocks of data is as an important error checking device in that a process that compares files may be led astray into finding and proceeding on the basis of a false match by the presence of blocks of redundant data, especially in the case text files and programs, both of which are known for containing blocks of redundant data.

The steps of the Source_FindSync Method are detailed the diagrammatic representation depicted in FIG. 9. Specifically, the Source_FindSync Method first checks the current Status to determine if it is set to Failure, in which case the Source_FindSync Method will be exited.

However, assuming that the current Status is not set to failure, the process will proceed to find a Small Match just past a given non-matching position using the Source FindMatch Method. In this case a Small Match means that a minimum patch size-type entry is provided for instances when looking at a buffer for a change in data, only one or two bytes of data have actually changed.

As seen in FIG. 10, the Source_FindMatch Method begins by initializing and normalizing the following internal values: (a) the setting of the Start Position; (b) the setting of the Buffer Positions and Lengths; (c) the Match Lengths; (d) the setting of Current Row to Start Position; and (e) the setting of End Row to the End of the Largest Buffer.

Next, a loop called the Loop for Comparing Data Rows is initiated. This loop begins by comparing the Current Row to see if it is smaller than the End Row. If
the Current row is greater than the End Row; the Loop for Comparing Data Rows thereby terminates, and subsequently, if there is a match (see FIG. 10A), then a Patch Action is written to skip the Older Version data (i.e., the data from the Older Version) and to insert the Newer Version data, after which the buffer positions are set past the Matching Data for the next Source_Read, then the Source FindMatch Method is exited and the status is returned. If however, no match is found, then the Source_FindMatch Method is directly exited and the status is returned.

Alternatively, if the Current Row in the Loop for Comparing Data Rows is less than the End Row, then a comparison is made to see if the End of the Newer Version is greater than the Current Row.

If the End of the Newer Row is greater the Current Row, then the Older Version of the Data Sets are compared to the Newer Version Data at the Current Row, and the End Compare is set to the smaller of the current row or the end of the Older Buffer (i.e., the buffer holding the Older Version data), and the Current Compare set is set to the first position (i.e., zero (0)). After completion of this, the process enters a Loop for Older Data Set Comparisons. The Loop for Older Data Set Comparisons Begins by comparing the Current Set to see if it is less than the End Set.

As seen in FIG. 8, if the Current Set is greater than the End Set, then the Loop is terminated, but if the Current Set is less than the End Set, then the current buffer set positions (i.e., values) are searched for a match position (i.e., the respective position numbers of the buffers that need to be compared are checked for identity). After implementation of the buffer values search above, then the two sets of data are checked for a match; if there is a match then the pointers are set to match positions, and the Loop for Older Data Set Comparisons is terminated; otherwise, if
there is no match then the current compare set is incremented and the Loop returned to find the proper match.

Once the Loop for Older Data Set Comparisons has terminated, if a match for the data set exists, then the Loop for Comparing Data Rows is terminated; but if no match exists, as seen in FIG. 10A, the Loop for Comparing data rows proceeds to compare the current row with the End of the Older Version, and to compare the data in the newer row. Should there be a match in this step, the Newer Version Data Set is then compared to the Older Version Data at the current row; the End Compare Set is set to the Smaller of the current row or the end of the newer buffer (i.e. set to one (1)), and the Current Compare Set is set to the first position (i.e., zero (0)). Once these comparisons have been accomplished, another loop called the Loop for Newer Data Set Comparisons is implemented. The Loop for Newer Data Set Comparisons begins with an assessment as to whether the Current Set is less than the End Set. If the Current Set is not less than the End Set, then the Loop for the Newer Data Set Comparisons is terminated. However, if the Current Set is less than the End Set, then the buffer values are searched (FIG. 8).

Once the buffer values search has been called and implemented, if a match has subsequently been found, then the pointers are set to the match positions, the Loop for Newer Data Set Comparisons is ended. Alternatively, if no subsequent match is found; then the current row is incremented and the process recommences the Loop for Comparing Newer Data Set Comparisons. Once the Loop for Newer Data Set Comparisons has terminated, then a determination is made as to whether a match has been found in the Newer Data Set; if no match has been found, then the Current Row is incremented and the Loop for Comparing Data Rows (FIG. 10) is recommenced. Once
a match has been found in the Newer Data Set, the Loop for Comparing Data Rows is terminated.

Once the Loop for Comparing Data Rows has been terminated, if there is a subsequent match found between the data rows, then the write Patch Action will skip the Older data (i.e., the Older Version data), insert the Newer data (i.e., the Newer Version data) and then set the buffer positions past the matching data for the next Source_Read. Once this has been accomplished, the Source_FindMatch Method is exited and the status is returned. Alternatively, if no subsequent match is found between the data rows, the Source_FindMatch Method is directly exited.

Once the Source_FindMatch Method is exited, the Source_FindSync Method continues (FIGS. 9-9A) by next determining whether a Small Match was found. If a Small Match has in fact been found, then the match and status are checked and depending on that particular outcome, the write patch action skips the Older Version data, inserts the Newer Version data and exits where the match and status are deemed correct, or if the match and status are not correct, it just directly exits.

If a Small Match is not found, then the File Buffers are checked for a match in size. If the File Buffer sizes match, then the Start Position is directly set past the data checked for the Small Match. If the sizes do not match, then Source_Read (FIGS. 6-6A) is called to increase the size of the smaller buffer. If by any chance the Source_Read fails, then the status is set to failure and the Source_FindSync Method is exited. Assuming, however, that there is no such failure, then the Start Position is set past the data checked for the small match. Similarly, should the file buffer sizes match, the Start Position will be set past the data checked for the small match.
After the Start Position has been set past the data checked for the small match, the process then enters a loop called a Loop for Finding Match. The first Step in the Loop for Finding Match involves calling the Source FindMatch Method (FIGS. 10-10A) with the start Position. After exiting the Source_Find Match Method, if a match is found, the Loop for Finding Match is terminated and then, if the match and Status are correct, the Write Patch Action is written to skip the Older data and insert the Newer data and the Source_FindSync Method is exited directly.

However, if a match is not found after exiting the Source_Find Match Method, then the Loop for Finding Match continues by increasing both file Buffers by 1 buffer by calling up the Source_Read Method (FIGS. 9 and 9A) and updating the start position to End of Largest Buffer Data. Having implementing the Source_Read Method if the Source_Read Method has failed the status will be set to failure and the Source_FindSync Method is exited. Alternatively, if the Source_Read Method has not failed, then there is a resulting assessment as to whether there are any buffer increases. If there have been any buffer increases, then the Loop for Finding a Match is re-implemented starting from the beginning of the Loop until there are no buffer increases. Once there are no buffer increases, the Loop for Finding Match is terminated.

After terminating the Loop for Finding a Match, the Match and Status are checked, and depending on this outcome, either the Patch Action will be written to skip Older Data and insert Newer Data, the Source_FindSync Method will be exited, or otherwise the Source_FindSync Method will just be directly exited.

Having exited the Source_FindSync Method, the process now returns to that shown on FIG. 8, where the status is set to current status and the Source_Compare Method is exited.
Returning to the series of 3 possible determinations made in the Source_Compare Method of FIG. 8, if alternatively, the original data size check should determine that the data size is not greater than zero (0), the next determination involves assessing whether the Older Buffer Length is greater than zero (0).

If the Older Buffer Length is greater than zero (0), then the end of the Older Version File does not exist in the Newer Version File and the Patch Action will be written to skip the Rest of the Older File and the Status will be set to the Current Status. Skipping the remainder of the Older File, just as in the skipping of any portion of the file, enables the process to save CPU cycles and utilize less memory and bandwidth. After executing these steps, the Source_Compare Method is thereby exited.

Alternatively, if the Older Buffer Length is not greater than zero (0), then the third possible determination in the Source_Compare Method is made. Specifically, the Newer Buffer Length is assessed as to whether it is greater than zero (0). If the Newer Buffer Length is greater than zero (0), then the End of the Newer Version File does not exist in the older file, and the Patch Actions will be written to skip the rest of the Older Version File and the Status will be set to the Current Status. After executing these steps, the Source Compare Method is thereafter exited.

However, should the Newer Buffer Length not be greater than zero (0), then the end of both files is reached and the status will be set to Failure. Once this point has been reached, then the Source_Compare Method is exited.

Referring now back to FIG. 3, once the Source Compare has been called and fully implemented, a determination is made as to whether the Source_Compare has succeeded. If the Source_Compare has succeeded, then
the Loop for Source Compare begins again from the beginning. In contrast, if the Source.Compare did not succeed, then Loop for Source.Compare is directly terminated.

5 Having completed the execution of the entire Loop for Source.Compare, the process now determines whether a buffer exists for the Older Version. If a buffer exists for the Older Version, free memory is then allocated for the Older Version. Having done this, the process proceeds to the next determinations involving the Newer Version. However, in the case where no buffer exists for the Older Version, the process proceeds directly to the next determination involving the Newer Version.

10 The next determination involving the Newer Version determines whether a buffer exists for the Newer Version. If such a buffer exists, then the free memory is allocated to the buffer, before calling the PatchOut Close Method to exit the PatchOut_Create Method. If no buffer exists for the Older Version, then the PatchOut Close Method is just called directly to exit the PatchOut Create Method.

15 As seen in FIG. 5, the PatchOut_Close Method first checks for a Failure status. If there is such a status, the PatchOut_Close Method is exited immediately. If no such Failure status exists, then the Internal Values for the Patch File Footer are set with Xor Check values and the CreateProcessStatus Virtual Method is notified. Thereafter, the Patch File Footer is written, and should this write fail, the status will be set to failure and the PatchOut_Close Method exited, but, if the write has not failed, the PatchOut_Close Method will be directly exited.

20 As seen at the bottom of FIG. 3, once the PatchOut_Close Method has been completed, the PatchOut Create Method is exited.
As seen in FIG. 2, having exited the PatchOut Create Method, the next step in the CreatePatchFile Method involves determining whether a File Pointer exists for the Older Version. The significance of a File Pointer is simply that by establishing an address for the Older Version, the process knows that the file has been opened.

Returning to the previous determination as to whether file pointers exist for all files, in the event that file pointers do not exist for all files, the process does not call the PatchOut_Create Method because if the process cannot create a file pointer, then the operating system was not able to access the files. Instead, the process merely proceeds directly to the next determination, i.e., whether a File Pointer exists for the Older Version.

Turning to the File Pointer determination for the Older Version, if no File Pointer exists for the Older Version, then the status is set to Failure and the process proceeds to the next step. However, if File Pointers do exist for the Older Version, then the Older Version File is closed and the process proceeds to the next step.

The next step in the process involves determining whether the file pointer exists for the Newer Version. If no file exists for the Older Version, then the process proceeds to a subsequent step. However, if File Pointers do exist for the Newer Version, then the Newer Version File is closed and the process proceeds to the next step.

A subsequent and final step in the Create Patch File Method is to determine whether a File Pointer exists for the Patch File. If no File Pointer exists for the Patch File, then the status is set to Failure and the Method concludes and the status is returned. Alternatively, if a File Pointer exists for the Patch
File, then the Patch File is closed, the Method is exited, and the status is returned.

Returning attention now to FIG. 1, once the Create Patch File Method has been fully implemented, the next broad step involves applying anywhere from zero to one or more of the Patch Files developed in the above portion(s) of the process.

Turning attention to FIGS. 11 and 12, the ApplyPatchFile Method (FIG. 11) involves a dependent Method, Patch File_Apply (FIG. 12). More specifically, the ApplyPatchFile is a Public Method which does the initial file checking and calls PatchFile_Apply, a Private Method in the present invention.

Thus, with specific reference to FIG. 11, the ApplyPatchFile Method begins by stacking the following File Name Pointers: (i) the Operating System (OS) File Name of the Older Version; (ii) the OS File Name of the Newer Version to Create; and (iii) the OS File Name of the Patch File to Apply.

Once the File Name Pointers have been properly stacked, the process will attempt to read the file size for the Older Version File. Should this Read Operation fail, the status will be set to Failure and the ApplyPatchFile Method will be exited.

However if the Read Operation does not fail, then the File Pointers are opened for all files. Having opened the File Pointers, a check is made to ensure that in fact all the File Pointers actually exist for all files. If the File Pointers exist for all files, then the Patch File_Apply Method is called.

The purpose of the Patch File_Apply Method is to distinguish the difference between two files (i.e., between the Older Version and the Newer Version). The PatchFile_Apply Method begins by performing the following initial steps: (i) initializing the Patch File Internals; (ii) reading the Patch File Header; (iii)
notifying the ApplyProcessStatus Virtual Method (this particular Method notifies the user of the Status); (iv) checking the Patch Process Version to the Process Version in the Headers; and (v) checking the File Size of the Passed File Name to Patch values in the Headers.

Having completed these initial steps, the next step is to determine if there are any Read errors or Version errors. Read errors are checked according to the status of the hardware (i.e., the memory); if there is a hardware failure, a Read error will be returned. Version errors are checked according to the file version name or denomination; if there is an inconsistency in the version of a file as read in the normalized file name, a version error will be returned. If there are any Read or Version errors, then the PatchFile_Apply Method is exited. If there are no Read or Version errors, then the PatchFile Apply Method continues and next initializes the Xor Check Values for both Older and Newer File Versions.

After initializing the Xor Check Values, a loop called the Loop for Patch File Read is utilized. The first step in the Loop for Patch File Read is to Read the Next Patch File Action Entry into the Internal Values, then return a Failure on the End of the File, and Read the Errors or the Invalid Data values.

Next, a determination is made as to whether there is a Read Failure or a Status Failure. Should there be such a Failure, the Loop for Patch File Read is ended. Alternatively, should there be no Read Failure or Status Failure, a loop nested written the Loop for Patch File Read, called the Loop for Skip to Start, is implemented.

The Loop for Skip to start begins by determining if the Older Version File Position is greater than or equal to the Action Start. If the Older Version File Position is greater than or equal to the Action Start, the Loop for Skip to start is ended, but if the
Older Version is not greater than or equal to the Action Start, the following sub-steps are taken: (i) copying the Older Version of the Data to the Newer Version; (ii) tracking the Xor Check and the file Position; and (iii) returning a Failure (if applicable) on the End of the File and reading errors or an invalid Xor Check Value (if any exist).

Once these three sub-steps have been implemented, then a determination is made as to whether the Copy failed; if the Copy has indeed failed, the Loop for Skip To Start is recommenced, otherwise, if the Copy has not failed, then the Loop for Skip To Start will conclude.

After the Loop for Skip To Start has been implemented, the next step in the progression involves determining whether the Older Version File Position is equal to the Action Start. If the Older Version File Position is not equal to the Action Start, then the status is set to Failure, and the Loop for Patch File Read is ended. If the Older Version File Position is equal to the Action Start then the following three sub-steps are implemented: (i) the Patch Action is handled, the Footer Xor values are checked and the New Data is inserted or the Older Data skipped; (ii) the ApplyProcessStatus Virtual Method of the Current Action is notified; and (iii) a failure is returned on either the Xor Check Failure or the Read errors.

Having completed these three sub-steps, a determination is made as to whether the Action has failed. If the Action has failed, then the Loop for Patch File Read is terminated, but if the Action has not failed, then the determination is made as to whether the Footer has been reached or the status is Failure. Should the Footer not have been reached or the status not be set to Failure, then the Loop for Patch File Read is recommenced; otherwise, if the Footers have been reached.
or the status is Failure, then the Loop for Patch File Read is terminated.

After the Loop for Patch File Read has concluded, the next to last determination in the PatchFile_Apply Method involves whether the status is Success but no Footer has been reached yet. The purpose of the Footer is to declare the patch actions and store the necessary RXC values for the comparisons used in error checking device steps. Normally the Footer will be reached at the end of the Patch File. If the Status is Success, but no Footer has been reached yet, the status is set to Failure and the PatchFile_Apply Method is exited. If the status is set to Success and the Footer has been reached, then the PatchFile_Apply Method is exited directly.

Having implemented the PatchFile_Apply Method, the remaining process in the Apply PatchFile Method to check if File Pointers exist for the Older Version. If File Pointers do not exist for the Older Version, then the Status is set to Failure and the process proceeds to the next file pointer check, which is a query as to whether File Pointers exist for the Newer Version. However, should File Pointers exist for the Older Version, the Older Version File is closed before heading to the check for Newer Version File Pointers.

Similar to the progression in the above-mentioned determination for Older Version Pointers, a determination is first made to assess whether File Pointers exist for the Newer Version. If no File Pointers exist for the Newer Version, the Status is set to Failure and proceeds to the next determination, however, if File Pointers do exist for the Newer Version, the Newer Version File is closed before proceeding to the next determination.

The next determination or check is for the existence of File Pointers for the Patch File. Analogous
to both of the immediately proceeding File Pointer determinations, if no File Pointers exist for the Patch File, then the status is set to Failure. If File Pointers exist for the Patch File, then the Patch File is closed.

The last step, in either case, is to exit the Apply Patch File Method and return the status.

The foregoing descriptions of preferred embodiments of the invention have been presented for purposes of illustration and description. They are not intended to be exhaustive or to limit the invention to the precise form disclosed. Obvious modifications or variations are possible in light of the above teachings. All such obvious modifications and variations are intended to be within the scope of the appended claims.
CLAIMS

We Claim:

1. A patching system for patching files in connected computer systems, said patching system comprising:

   means for reading a Newer Version File in a reference computer and a corresponding Older Version File in a reference computer;

   comparison means for determining which of the corresponding Older Version Files in a computer system need to be patched;

   means for writing a Patch File containing at least one Patch Action based on the results of said comparison;

   means for applying said Patch File to an Older Version File in a target computer system to create a Patched Version File;

   an error checker for checking for any differences between the reference Newer Version File and the reference Older Version File, and for checking any differences between the Patched Version File and the Newer Version File.

2. The system according to Claim 1 wherein the error checker comprises a 32 bit RXC algorithm.

3. The system according to Claim 2 wherein the Patch File is compressed for transmitting from the reference computer system to the target computer system.

4. The system according to Claim 3 wherein the Patch File is encrypted for transmitting from the reference computer system to the target computer system.
5. The system according to Claim 1 wherein the comparison means includes the use of an enhanced pyramidal scheme.

6. The system according to Claim 1 wherein the means for writing a Patch File includes the use of an efficient patch action allocation scheme.

7. A patching system for patching files in computer systems, said patching system comprising:
   a reference computer;
   a target computer coupled to said reference computer via a TCP/IP connection;
   a memory portion, coupled to at least one said computer; said memory portion comprising:
   software for reading a reference Newer Version File and a reference Older Version File;
   software for comparing the reference Newer Version File with the reference Older Version File in order to create a Patch File containing at least one Patch Action;
   software for applying the Patch File to a target Older Version File in order to create a Patched Version File; and
   software for error checking the above steps in the patching system.

8. The system according to Claim 7 wherein the software for error checking uses a 32 bit RXC algorithm.

9. The system according to Claim 8 wherein the software for applying the Patch File to a target Older Version File Patch File includes a means for compression of the Patch File when transmitting the Patch File from the reference computer system to the target computer system.
10. The system according to Claim 9 wherein the software for applying the Patch File to a target Older Version File Patch File includes a means for encryption of the Patch File when transmitting the Patch File from the reference computer system to the target computer system.

11. The system according to Claim 7 wherein the software for comparing includes the use of an enhanced pyramidal scheme.

12. The system according to Claim 7 wherein the means for comparing a Patch File includes the use of an efficient patch action allocation scheme.

13. A method of patching files in remote computer systems, comprising the steps of:
   accessing a reference Newer Version File in a reference computer system;
   accessing a corresponding reference Older Version File in the reference computer system;
   error checking the correspondence of the reference Older Version File with the reference Newer Version File;
   comparing the reference Older Version File with the reference Newer Version File in order to prepare a Patch File having at least one patch action;
   applying the Patch File to a target Older Version to create a Patched Version File; and
   error checking the Patched Version with the Newer Version.

14. The method of patching files according to Claim 13 wherein the error checking comprises a 32 bit RXC algorithm.
15. The method of patching files according to Claim 14 wherein the step applying the Patch File to a target Older Version File Patch File includes a means for compression of the Patch File when transmitting the Patch File from the reference computer system to the target computer system.

16. The method of patching files according to Claim 15 wherein the step of applying the Patch File to a target Older Version File Patch File includes a means for encryption of the Patch File when transmitting the Patch File from the reference computer system to the target computer system.

17. The method of patching files according to Claim 13 wherein the comparing step includes the use of an enhanced pyramidal scheme.

18. The method of patching files according to Claim 13 wherein the comparing step includes the use of an efficient patch action allocation scheme.

19. A file patcher for patching files in connected computer systems, said file patcher comprising:

   means for reading a reference Newer Version File on a memory location of a reference computer;

   means for reading a corresponding reference Older Version File in the memory location of the reference computer;

   means for comparing the reference Older Version File with the reference Newer Version File for purposes of preparing a Patch File having at least one Patch Action;
means of applying the Patch File to a corresponding Older Version in the memory of a target computer;

means of error checking the comparing means and the applying means so as to reduce the likelihood of incurring patch errors.

20. The file patcher according to Claim 19 wherein the means of error checking comprises a 32 bit RXC algorithm.

21. The file patcher according to Claim 20 wherein the means for applying the Patch File to a target Older Version File Patch File includes a means for compression of the Patch File when transmitting the Patch File from the reference computer system to the target computer system.

22. The file patcher according to Claim 21 wherein the means for applying the Patch File to a target Older Version File Patch File includes a means for encryption of the Patch File when transmitting the Patch File from the reference computer system to the target computer system.

23. The file patcher according to Claim 19 wherein the means for comparing includes the use of an enhanced pyramidal scheme.

24. The file patcher according to Claim 19 wherein the means for comparing includes the use of an efficient patch action allocation scheme.
25. A file patcher stored via storage media for updating computer system files, said storage media comprising:

   a first plurality of binary values for reading a reference Newer Version File on a reference computer;

   a second plurality of binary values for reading a corresponding reference Older Version File on a reference computer;

   a third plurality of binary values for comparing the reference Newer Version File with corresponding reference Older Version File so as to Create a Patch File having at least one Patch Action;

   a fourth plurality of binary values for applying said Patch File to a corresponding Older Version File on a target computer to create a Patched Version File;

   a fifth plurality of binary values for preventing errors by the file patcher.

26. The file patcher stored via storage media according to Claim 25 wherein the plurality of binary values for preventing errors further comprises a plurality of binary values for implementing a 32 bit RXC algorithm.

27. The file patcher stored via storage media according to Claim 26 wherein the fourth plurality of binary values for applying the Patch File to a target Older Version File Patch File includes a means for compression of the Patch File when transmitting the Patch File from the reference computer system to the target computer system.

28. The file patcher stored via storage media according to Claim 27 wherein the fourth plurality of
binary values for applying the Patch File to a target Older Version File. Patch File includes a means for encryption of the Patch File when transmitting the Patch File from the reference computer system to the target computer system.

29. The file patcher according to Claim 25 wherein the third plurality of binary values for comparing includes the use of an enhanced pyramidal scheme.

30. The file patcher according to Claim 25 wherein the third plurality of binary values for comparing includes the use of an efficient patch action allocation scheme.

31. A method of manufacturing data management systems comprising the steps of:
   reading a reference Newer Version File of a reference computer system;
   reading a corresponding reference Older Version File of a reference computer;
   comparing the reference Older Version File with the reference Newer Version File in order to prepare a Patch File having at least one Patch Action applying the Patch File to the corresponding target Older Version File of a target computer to create a Patched Version File;
   error checking the above method to minimize the likelihood of errors.

32. The method of manufacturing data management systems according to Claim 31 wherein error checking step further comprises the steps of utilizing a 32 bit RXC algorithm.
33. The method of manufacturing data management systems according to Claim 32 wherein the step of applying the Patch File to a target Older Version File Patch File includes a means for compression of the Patch File when transmitting the Patch File from the reference computer system to the target computer system.

34. The method of manufacturing data management systems according to Claim 33 wherein the step of applying the Patch File to a target Older Version File Patch File includes a means for encryption of the Patch File when transmitting the Patch File from the reference computer system to the target computer system.

35. The method of manufacturing according to Claim 31 wherein the step of comparing includes the use of an enhanced pyramidal scheme.

36. The method of manufacturing according to Claim 31 wherein the step of comparing includes the use of an efficient patch action allocation scheme.

37. An object oriented data patching data structure stored in a memory for access when patching files in a remote computer system, comprising:

   an accessing method on a reference computer, said accessing method having a Read Method Stored therein for accessing a reference Newer Version File and corresponding reference Older Version File;

   a comparing method for comparing the reference Newer Version File and corresponding reference Older Version File for creating a Patch File data structure having at least one set of data representing at least one Patch Action;
an application method for applying the Patch File data to a corresponding target Older Version File in order to create a Patched Version File;

an error checking method for minimizing the likelihood of errors in implementing the comparing and application methods.

38. The object oriented data patching data structure according to Claim 37 wherein the error checking method comprises a 32 bit RXC algorithm.

39. The object oriented data patching data structure according to Claim 38 wherein the application method for applying the Patch File to a target Older Version File Patch File includes a means for compression of the Patch File when transmitting the Patch File from the reference computer system to the target computer system.

40. The object oriented data patching data structure according to Claim 39 wherein the application method for applying the Patch File to a target Older Version File Patch File includes a means for encryption of the Patch File when transmitting the Patch File from the reference computer system to the target computer system.

41. The object oriented data patching data structure according to Claim 37 wherein the method of comparing includes the use of an enhanced pyramidal scheme.

42. The object oriented data patching data structure according to Claim 37 wherein the method of comparing includes the use of an efficient patch action allocation scheme.
43. A computer data signal embodied in a carrier wave for file updating in computer systems comprising:
   a source code segment for reading a reference Newer Version File on a reference computer;
   a source code segment for reading a corresponding reference Older Version File on a reference computer;
   a source code segment for comparing the reference Newer Version File with corresponding reference Older Version File so as to create a Patch File having at least one Patch Action;
   a source code segment for applying said Patch File to a corresponding Older Version File on a target computer to create a Patched Version File;
   a source code segment for preventing errors in a file patcher during the file updating.

44. The computer data signal embodied in a carrier wave for file updating in computer systems according to Claim 43 wherein the source code segment for preventing errors further comprises a plurality of binary values for implementing a 32 bit RSC algorithm.

45. The computer data signal embodied in a carrier wave for file updating in computer systems according to Claim 44 wherein the source code segment for applying the Patch File to a target Older Version File Patch File includes a means for compression of the Patch File when transmitting the Patch File from the reference computer system to the target computer system.

46. The computer data signal embodied in a carrier wave for file updating in computer systems according to Claim 45 wherein the source code segment for applying the Patch File to a target Older Version File
Patch File includes a means for encryption of the Patch File when transmitting the Patch File from the reference computer system to the target computer system.

47. The computer data signal embodied in a carrier wave for file updating in computer systems according to Claim 43 wherein the source code segment for comparing includes the use of an enhanced pyramidal scheme.

48. The computer data signal embodied in a carrier wave for file updating in computer systems according to Claim 43 wherein the source code segment for comparing includes the use of an efficient patch action allocation scheme.
FilePatchC Class FlowChart: Usage, Constructor, and Destructor

FilePatchC Usage

Create Instance of FilePatch C with Constructor Method → FilePatchC Constructor

Create One or More Patch Files or None And/Or Apply One or More Patch Files, or None

Release Instance of FilePatchC with Destructor Method → FilePatchC Destructor

FilePatchC Constructor Method

Allocate Memory for Class Structure and Virtual Method Pointers

Assign and Normalize Initial File Patch Parameters

prmBufferSize : Comparison Buffer Size
prmMissBufs : Maximum Buffers Loaded when Searching for Next Match
prmXrcSpacing : Output Xor Check when this Number of Bytes Processed
prmMatchMin : Minimum Number of Bytes determining a Found Match
prmSmallBuf : Size of Data to Check for a Small Difference in Versions
prmSmallMin : Minimum Number of Bytes to Match for a Small Difference

Exit FilePatchC Constructor Method

FilePatchC Destructor Method

Free Memory Allocated for Class Structure and Virtual Method Pointers

Exit FilePatchC Destructor Method

FIG. 1
3/15

FilePatchC Class FlowChart: CreatePatchFile Method

CreatePatchFile Method

Stack the FileName Pointers
- OS File Name of the Older Version
- OS File Name of the Newer Version
- OS File Name of the Patch File to Create

Attempt to Read File Sizes for the Older and Newer Versions

Read Failed? YES NO

Set Status to Failure

Exit CreatePatchFile Method

Open File Pointers to All Files

Do File Pointers Exist for All Files?

YES NO

Call PatchOut_Create Method

PatchOut_Create

Does File Pointer Exist for Older Version?

NO YES

Close Older Version File

Set Status to Failure

Does File Pointer Exist for Newer Version?

NO YES

Close Newer Version File

Set Status to Failure

Does File Pointer Exist for Patch File?

NO YES

Close Patch File

Set Status to Failure

Exit CreatePatchFile Method and Return Status

FIG. 2

SUBSTITUTE SHEET (RULE 26)
**FilePatchC Class FlowChart: PatchOut_Create Method**

**PatchOut_Create Method**

- Call PatchOut_Init Method

  **PatchOut_Init**
  
  **PatchOut_Init Failed?**
  
  YES → Set Status to Failure
  
  **Exit PatchOut_Create Method**

  **Initialize Buffer Pointers to NULL and Buffer Position to 0 (First Position)**
  
  **Initialize Xor Check Values for Both Versions**

**Look for Source Compare**

- Call Source_Read Method for Older Version
  
  **Source_Read (Older)**

- Call Source_Read Method for Newer Version
  
  **Source_Read (Newer)**

- Call Source_Compare Method
  
  **Source_Compare**

  **Source_Compare Succeed?**

  YES → End Loop for Source Compare

  **NO**

**Does Buffer Exist for Older Version?**

**YES**

- Free Memory Allocated to Buffer for Older Version

**NO**

**Does Buffer Exist for Newer Version?**

**YES**

- Free Memory Allocated to Buffer for Older Version

**NO**

**Exit PatchOut_Create Method**

**FIG. 3**

SUBSTITUTE SHEET (RULE 26)
**FilePatchC Class FlowChart: PatchOut_Create Method**

1. Call **PatchOut_Init Method**
   - Set Internal Values for the Patch File Header and Notify the CreateProcessStatus Virtual Method
   - Write the Patch File Header
   - **NO** Write Failed? **YES** Set Status to Failure
     - **Exit PatchOut_Init Method**
   - Set Status to Success
   - **Exit PatchOut_Init Method**

**FIG. 4**

**PatchOut_Close Method**

1. **NO** Status indicates Failure? **YES** "Exit PatchOut_Close Method"
   - Set Internal Values for the Patch File Footer with Xor Check Values and Notify the CreateProcessStatus Virtual Method
   - Write the Patch File Header
   - **NO** Write Failed? **YES** Set Status to Failure
   - **Exit PatchOut_Close Method**

**FIG. 5**

SUBSTITUTE SHEET (RULE 26)
FilePatchC Class FlowChart: Source_Read Method

Source_Read Method

Pull Source_Read Parameters from Stack
  num : File Buffer to Read (Older/Newer)
  addBuf: (Defaults to 0) Additional Buffers to Create on this Read

YES
File Buffer Exists?

NO

Reset Additional Buffers to Zero (0)
Set Number of Existing Buffers to One (1)
Determine Buffer Size Parameters
Allocate Buffer memory and Assign Buffer Pointer

YES
File Buffer Exists?

NO
Set Status to Failure
Exit Source_Read Method

Reset Buffer Data Values to Empty (0)

NO
Any Additional Buffers to Add?

YES
"Additional Buffer" Branch

Increment Count of Buffers
Normalized Count to Maximum Buffers Allowed
Determine Buffer Size Parameters
ReAllocate Buffer Memory and ReAssign Buffer Pointer

YES
File Buffer Exists?

NO
Set Status to Failure
Exit Source_Read Method

Reset Buffer Data Keep Position to First Position (0)

NO "Additional Buffer" Branch

End of "NO Additional Buffer" Branch
(Fig. 6A)

NO

0<Buffer Data Keep < Buffer Length

YES

Subtract Keep Position from Buffer Length
Move Data at Keep Position to Beginning of Buffer
Increment Current File Position to Keep Position

Increment Current File Position to End of Buffer (Length)
Reset Buffer Empty (0)

Continue "NO Additional Buffer" Branch in Fig. 6a

FIG. 6

SUBSTITUTE SHEET (RULE 26)
FilePatchC Class FlowChart: Source_Read Method

Continue "NO Additional Buffer" Branch from Fig. 6

Reset Buffer Data Keep Position to First Position (0)

NO

More Than One Buffer?

YES

Determine Necessary Buffer Amount from Current Data in Buffer

NO

Necessary Buffers < Current Buffers

YES

Reset Buffer Size to Necessary Size
ReAllocate Buffer Memory and ReSet Buffer Pointer

File Buffer Exists

NO

Set Status to Failure
Exit Source_Read Method

YES

"Additional Buffer" Branch Continues Here (From FIG. 6)

End of "NO Additional Buffer" Branch

NO

Data Length < Buffer Size?

YES

Block Read Data to Fill Empty Area of File Buffer
Notify CreateProcessStatus Virtual Method

NO

Read Less Than Full Buffer?

YES

Did IO Error Occur?

NO

Loop Through New Data

YES

Call SourceRxc_AddData to Add Values to Xor Check

End of New Data?

NO

End Loop Through New Data

YES

Exit Source_Read Method

Exit Source_Read Method and Return Status

FIG. 6A
FilePatchC Class FlowChart: SourceRxc_AddData Method

SourceRxc_AddData Method

Pull Buffer Number and Data from Stack
RXC = ((RXC ShiftRight 8 bits) OR (RXC ShiftLeft 24bits)) XOR Data
Increment Current File Position Counter

NO

Reached File Check Position?
(for PatchApply)

YES

NO

RXC Dosen't Match?

YES

Set Status to RXC Failure

Reset File Check Position

ExitSourceRxc_AddData Method

FIG. 7
**FilePatchC Class FlowChart: Source_Compare Method**

**Source_Compare Method**

- Current Status Set to Failure?
  - NO
  - Set the Comparison Data Size to the Smallest File Buffer Size
  - NO
  - Data Size > 0?
    - YES
    - Set Index to Position of First Non-Matching Byte in Buffers or Buffer Size
    - NO
    - Index > 0?
      - YES
      - Set Internal Values to Skip these Matches during Next Source_Read
      - Set Status to Success and Exit
    - NO
    - Exit Source_Compare Method
  - YES
  - Call Source_FindSync Method → **Source_FindSync**
    - Set Status to Current Status → ExitSource_Compare Method
  - NO
  - Older Buffer Length > 0?
    - YES
    - End of Older File Does Not Exist in Newer File:
      - Write Patch Action to Skip the Rest of the Older File
      - Set Status to Current Status
      - ExitSource_Compare Method
    - NO
      - Newer Buffer Length > 0?
        - YES
        - End of Newer File Does Not Exist in Older File:
          - Write Patch Action to Skip the Rest of the Newer File
          - Set Status to Current Status
          - ExitSource_Compare Method
        - NO
          - End of Both Files Reached:
            - Set Status to Failure
            - ExitSource_Compare Method
- Exit Source_Compare Method and Return Status

**FIG. 8**

*SUBSTITUTE SHEET (RULE 26)*
FilePatchC Class FlowChart: Source_FindSync Method

**Source_FindSync Method**

- **NO** Current Status Set to Failure? → **YES** Set Status to Failure → **Exit Source_FindSync Method**
  
  **Find a Small Match Just Past Non-Matching Position** → Source_FindMatch

  **YES** Small Match Found → **NO**
  
  **YES** Do File Buffer Sizes Match? → **NO**
  
  **Call Source_Read to Increase Small Buffer** → Source_Read

  **NO** Source_Read Failure? → **YES** Set Status to Failure → **Exit Source_FindSync Method**

  **Set Start Position past Data Checked for the Small Match**

  **Loop for Finding Match**

  **Call Source_FindMatch with Start Position** → Source_FindMatch

  **NO** Match Found? → **YES**

  **Increase Both File Buffers by 1 Buffer by Calling Source_Read for Both Buffers Update Start Position to End of Largest Buffer Data** → Source_Read

  **NO** Source_Read Failure? → **YES** Set Status to Failure → **Exit Source_FindSync Method**

  **YES** Any Buffer Increases? → **NO**

  **End Loop for Finding Match**

  **NO** No Match and Status OK? → **YES**

  Write Patch Action to Skip Older Data and Insert Newer Data

**Exit Source_FindSync Method and Return Status**

**FIG. 9**
11/15

**FilePatchC Class FlowChart: Source_FindMatch Method**

**Source_FindMatch Method**

- Initialize and Normalize Internal Values
  - Set Start Position
  - Set Buffer Positions and Lengths
  - Set Match Lengths
  - Set Current Row to Start Position
  - Set End Row to End of Largest Buffer

**Loop for Comparing Data Rows**

- **YES**: Current Row < End Row
- **NO**: End Loop for Comparing Data Rows (FIG. 10A)

**FIG. 10**

- Comparing Older Version Data Sets to Newer Version Data at Current Row
  - Set End Compare Set to Smaller of (Current Row, End of Older Buffer)
  - Set Current Compare Set to 0 (First Position)

**Loop for Older Data Set Comparisons**

- **YES**: Current Set < End Set
- **NO**: Search Current Buffer Set Positions for Match

- **NO**: Match Found
- **YES**: Set Pointers to Match Positions
  - Increment Current Compare Set
  - End Loop for Older Data Set Comparisons

**End Loop for Comparing Data Rows (FIG. 10A)**

**Continue Loop for Comparing Data Rows (FIG. 10A)**
FilePatchC Class FlowChart: Source_FindMatch Method (Continued from FIG. 10)

Continue Loop for Comparing Data Rows (From FIG. 10)

- NO
- (Current Row < End Older) AND Data in Newer

Comparing Newer Version Data Sets to Older Version Data at Current Row
Set End Compare Set to Smaller of (Current Row, End of Newer Buffer-1)
Set Current Compare Set to 0 (First Position)

Loop for Newer Data Set Comparisons

- YES
- Current Set < End Set

Search Current Buffer Set Positions for Match

- NO
- Match Found

Set Pointers to Match Positions
Increment Current Compare Set

End Loop for Newer Data Set Comparisons

- YES
- NO
- Match Found

Increment Current Row

Loop for Comparing Data Rows (FIG. 10)

End Loop for Comparing Data Rows

- NO
- Match Found

Write Patch Action to Skip Older Data and Insert Newer Data
Set Buffer Positions Past Matching Data for Next Source_Read

Exit Source_FindMatch Method and Return Status

FIG. 10A
FilePatchC Class FlowChart: ApplyPatchFile Method

ApplyPatchFile Method

Stack the FileName Pointers
- OS File Name of the Older Version
- OS File Name of the Newer Version to Create
- OS File Name of the Patch File to Apply

Attempt to Read File Size for the Older Version File

Read Failed?
- YES: Set Status to Failure
- NO: Exit ApplyPatchFile Method

Open File Pointers to All Files

Do File Pointers Exist for All Files?
- YES: Call PatchFile_Apply Method
- NO: Exit ApplyPatchFile Method

PatchFile_Apply

Does File Pointer Exist for Older Version?
- YES: Close Older Version File
- NO: Set Status to Failure

Does File Pointer Exist for Newer Version?
- YES: Close Newer Version File
- NO: Set Status to Failure

Does File Pointer Exist for Patch File?
- YES: Close Patch File
- NO: Set Status to Failure

Exit ApplyPatchFile Method and Return Status

FIG. 11

SUBSTITUTE SHEET (RULE 26)
FilePatchC Class FlowChart: PatchFile_Apply Method

PatchFile_Apply Method

Initialize PatchFile Internals
Read Patch File Header
Notify ApplyProcessStatus Virtual Method
Check Patch Process Version to Process Version in Headers
Check Passe File Name its File Size to Patch Values in Headers

Initialize Xor Check Values for Both File Versions

Loop for Patch File Read

Read Next Patch FileAction Entry into Internal Values
Return Failure on End of File, Read Errors or Invalid Data Values

NO

YES

Read Failure Or Status Failure ?

Loop for Skip to Start

NO

Older Version File Position >= Action Start?

YES

Copy Older Version Data to Newer Version
Track Xor Check and File Position
Return Failure on End of File, Read Errors, or Invalid Xor Check Value

NO

YES

Copy Failed?

End Loop for Skip to Start

FIG. 12A

FIG. 12

SUBSTITUTE SHEET (RULE 26)
Older Version File Position $\geq$ Action Start?

Set Status to Failure

Handle Action: Check any RXC, or Insert New Data, or Skip old Data or check Footer RXCs
Notify ApplyProcessStatus Virtual method of CurrentAction
Return Failure on Xor Check Failure or Read Errors

Action Failed?

YES

NO

Footer Reached OR Failure Status?

YES

End Loop for Patch File Read

NO

Success, But No Footer Yet?

YES

Set Status to Failure

Exit PatchFile_Apply Method

FIG. 12B
INTERNATIONAL SEARCH REPORT

A. CLASSIFICATION OF SUBJECT MATTER

IPC(6) : G06P 17/30
US CL. : 707/203

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)

U.S. : 707/203; 707/200-204; 714/48, 52; 341/94, 104-106

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 practicable, search terms used)

APS, IEEE, NPL, WEST 1.2

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>
</table>

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

* Special categories of cited documents:
  *A* document defining the general state of the art which is not considered to be of particular relevance
  *E* earlier document published on or after the international filing date
  *L* document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
  *O* document referring to an oral disclosure, use, exhibition or other means
  *P* document published prior to the international filing date but later than the priority date claimed

* T* later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

* X* document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

* Y* document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is combined with one or more other such documents, each combination being obvious to a person skilled in the art

* A* document member of the same patent family

Date of the actual completion of the international search: 25 SEPTEMBER 1999

Date of mailing of the international search report: 18 OCT 1999

Name and mailing address of the ISA/US Commissioner of Patents and Trademarks
Box PCT
Washington, D.C. 20231
Facsimile No. (703) 305-3230

Authorized officer: HOSAIN T. ALAM
Telephone No. (703) 308-6662

Form PCT/ISA/210 (second sheet)(July 1992)