(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
4 December 2008 (04.12.2008)

PCT

(10) International Publication Number
WO 2008/148076 A1

(51) International Patent Classification:
G06F 9/46 (2006.01)

(21) International Application Number:
PCT/US2008/064812

(22) International Filing Date: 25 May 2008 (25.05.2008)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
11/807,130 25 May 2007 (25.05.2007) US

(71) Applicant *(for all designated States except US)*: **MICROSOFT CORPORATION** [US/US]; One Microsoft Way, Redmond, Washington 98052-6399 (US).

(72) Inventors: **WANG, Lifeng**; One Microsoft Way, Redmond, Washington 98052-6399 (US). **WANG, Jian**; One Microsoft Way, Redmond, Washington 98052-6399 (US). **LI, Yang**; One Microsoft Way, Redmond, Washington 98052-6399 (US). **LIU, Yunxin**; One Microsoft Way, Redmond, Washington 98052-6399 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RS, RU, SC, SD, SE, SG, SK, SL, SM, SV, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MT, NL,

[Continued on next page]

(54) Title: LAZY KERNEL THREAD BINDING

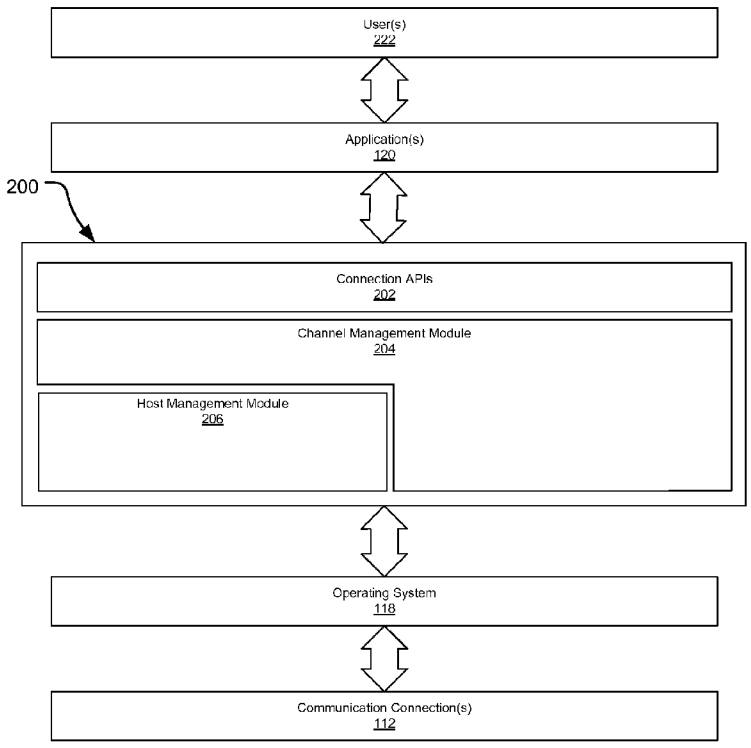

Fig. 2

(57) Abstract: Various technologies and techniques are disclosed for providing lazy kernel thread binding. User mode and kernel mode portions of thread scheduling are decoupled so that a particular user mode thread can be run on any one of multiple kernel mode threads. A dedicated backing thread is used whenever a user mode thread wants to perform an operation that could affect the kernel mode thread, such as a system call. For example, a notice is received that a particular user mode thread running on a particular kernel mode thread wants to make a system call. A dedicated backing thread that has been assigned to the particular user mode thread is woken. State is shuffled from the user mode thread to the dedicated backing thread using a state shuffling process. The particular kernel mode thread is put to sleep. The system call is executed using the dedicated backing thread.

WO 2008/148076 A1

NO, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Declarations under Rule 4.17:

— *as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii))*

— *as to the applicant's entitlement to claim the priority of the earlier application (Rule 4.17(iii))*

Published:

— *with international search report*

LAZY KERNEL THREAD BINDING

BACKGROUND

[001] Over time, computer hardware has become faster and more powerful. For example, computers of today can have multiple processor cores that can operate in parallel. Programmers would like for different pieces of the program to execute in parallel on these multiple processor cores to take advantage of the performance improvements that can be achieved. A high performance parallel application must exert careful control over its execution to optimize the use of hardware caches and interconnects.

[002] However, operating systems of today are limited in their ability to allow applications to control scheduling of their threads. For example, some operating systems of today, such as MICROSOFT® WINDOWS® support two ways for allowing applications to schedule their own execution. The first way is that an application can adjust its thread state (runnable or suspended), the thread priority, etc. However, the time it takes to put one thread to sleep and start another one using this approach is relatively expensive. Furthermore, a user mode thread can only execute on its associated kernel thread. This makes it difficult to use threads to control application execution in parallel and/or other applications.

[003] The second way an application can schedule its own execution is to use fibers. A fiber is a lightweight execution context that can be scheduled entirely in user mode. However, most operating system services are built around threads as opposed to fibers, and these system services are hard to use or do not work at all when called from fibers. Thus, fibers are also difficult to use in controlling application execution in parallel and/or other operations.

SUMMARY

[001] Various technologies and techniques are disclosed for providing lazy kernel thread binding. User mode and kernel mode portions of thread scheduling are decoupled so that a particular user mode thread can be run on any one of multiple kernel mode threads. In one implementation, each user mode thread is given a dedicated backing thread. A respective dedicated backing thread is used whenever a user mode thread wants to perform an operation that could affect the kernel mode thread, such as a system call. For example, a notice is received that a particular user mode thread running on a particular kernel mode thread wants to make a system call. A dedicated backing thread that has been assigned to the particular user mode thread is woken. State is shuffled from the user mode thread to the dedicated backing thread using a state shuffling process.

[002] In one implementation, the state shuffling process begins upon receiving notice that a particular user mode thread running on a particular kernel mode thread wants to make a system call. A register state of the particular user mode thread is saved. The particular kernel mode thread is put to sleep. A respective backing thread is woken that was assigned to the particular user-mode thread. The register state is restored to the respective backing thread. The system call is executed using the dedicated backing thread.

[003] This Summary was provided to introduce a selection of concepts in a simplified form that are further described below in the Detailed Description. This Summary is not intended to identify key features or essential features of the claimed subject matter, nor is it intended to be used as an aid in determining the scope of the claimed subject matter.

BRIEF DESCRIPTION OF THE DRAWINGS

[004] Figure 1 is a diagrammatic view of a computer system of one implementation.

[005] Figure 2 is a diagrammatic view of a lazy kernel thread binding application of one implementation operating on the computer system of Figure 1.

[006] Figure 3 is a high-level process flow diagram for one implementation of the system of Figure 1.

[007] Figure 4 is a process flow diagram for one implementation of the system of Figure 1 illustrating the more detailed stages involved in using dedicated backing threads for system calls for a user mode thread running on a kernel mode thread.

[008] Figure 5 is a process flow diagram for one implementation of the system of Figure 1 illustrating the stages involved in shuffling state from a user mode thread running on a kernel mode thread to a backing thread.

[009] Figure 6 is diagram illustrating how a user mode thread running on a kernel mode thread is transitioned to a dedicated backing thread when a system call is made.

[010] Figure 7 is a process flow diagram for one implementation of the system of Figure 1 that illustrates the stages involved in handling subsequent user mode thread executions.

[011] Figure 8 is a process flow diagram for one implementation of the system of Figure 1 that illustrates the stages involved in providing thread affinity when using lazy kernel thread binding.

DETAILED DESCRIPTION

[012] For the purposes of promoting an understanding of the principles of the invention, reference will now be made to the embodiments illustrated in the drawings and specific language will be used to describe the same. It will nevertheless be understood that no limitation of the scope is thereby intended. Any alterations and further modifications in the described embodiments, and any further applications of the principles as described herein are contemplated as would normally occur to one skilled in the art.

[013] The system may be described in the general context as an application that enhances operating system thread scheduling, but the system also serves other purposes in addition to these. In one implementation, one or more of the techniques described herein can be implemented as features within an operating system program such as MICROSOFT® WINDOWS®, or from any other type of program or service that manages and/or executes threads.

[014] In one implementation, a system is provided that decouples user mode and kernel mode portions of thread scheduling so that a particular user mode thread can be run on any one of multiple kernel mode threads. Each user mode thread is assigned a respective dedicated backing thread. A respective dedicated backing thread is used whenever a particular user mode thread wants to perform an operation that could affect the kernel mode thread, such as a system call. A state shuffling process is used to shuffle state from the user mode thread running on the kernel mode thread to the dedicated backing thread, and then the dedicated backing thread is then used to make the system call.

[015] As shown in Figure 1, an exemplary computer system to use for implementing one or more parts of the system includes a computing device, such as computing device 100. In its most basic configuration, computing device 100 typically includes at least one processing unit 102 and memory 104. Depending on the exact configuration and type of computing device, memory 104 may be volatile (such as RAM), non-volatile (such as ROM, flash memory, etc.) or some combination of the two. This most basic configuration is illustrated in Figure 1 by dashed line 106.

[016] Additionally, device 100 may also have additional features/functionality. For example, device 100 may also include additional storage (removable and/or non-removable) including, but not limited to, magnetic or optical disks or tape. Such additional storage is illustrated in Figure 1 by removable storage 108 and non-removable storage 110. Computer storage media includes volatile and nonvolatile, removable and non-removable media implemented in any method or technology for storage of information such as computer readable instructions, data structures, program modules or other data. Memory 104, removable storage 108 and non-removable storage 110 are all examples of computer storage media. Computer storage media includes, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, CD-ROM, digital versatile disks (DVD) or other optical storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium which can be used to store the desired information and which can accessed by device 100. Any such computer storage media may be part of device 100.

[017] Computing device 100 includes one or more communication connections 114 that allow computing device 100 to communicate with other computers/applications 115. Device 100 may also have input device(s) 112 such as keyboard, mouse, pen, voice input device, touch input device, etc. Output device(s) 111 such as a display, speakers, printer, etc. may also be included. These devices are well known in the art and need not be discussed at length here. In one implementation, computing device 100 includes lazy kernel thread binding application 200. In one implementation, lazy kernel thread binding application can be part of an operating system executing on computing device 100 or some other application. Lazy kernel thread binding application 200 will be described in further detail in Figure 2.

[018] Turning now to Figure 2 with continued reference to Figure 1, a lazy kernel thread binding application 200 operating on computing device 100 is illustrated. Lazy kernel thread binding application 200 is one of the application programs that reside on computing device 100. However, it will be understood that lazy kernel thread binding application 200 can alternatively or additionally be embodied as computer-executable instructions on one or more computers and/or in different variations than shown on Figure 1. Alternatively or additionally, one or more parts of lazy kernel thread binding application 200 can be part of system memory 104, on other computers and/or applications 115, or other such variations as would occur to one in the computer software art.

[019] Lazy kernel thread binding application 200 includes program logic 204, which is responsible for carrying out some or all of the techniques described herein. Program logic 204 includes logic for decoupling user mode and kernel

mode portions of thread scheduling so that a particular user mode thread can be run on any one of a plurality of kernel mode threads 206; logic for moving a user mode thread running on a kernel mode thread to a dedicated backing thread when the user mode thread wants to perform an action that could affect the kernel mode thread (e.g. system calls, etc.) 208; logic for providing a user mode scheduler that is responsible for dispatching the particular user mode thread on a particular kernel mode thread 210; logic for providing thread affinity 212; and other logic for operating the application 220. In one implementation, program logic 204 is operable to be called programmatically from another program, such as using a single call to a procedure in program logic 204.

[020] Turning now to Figures 3-8 with continued reference to Figures 1-2, the stages for implementing one or more implementations of lazy kernel thread binding application 200 are described in further detail. Figure 3 is a high level process flow diagram for lazy kernel thread binding application 200. In one form, the process of Figure 3 is at least partially implemented in the operating logic of computing device 100. The process begins at start point 240 with giving each user mode thread a dedicated backing thread (stage 242). In one implementation, the dedicated backing thread is also a kernel mode thread. These dedicated backing threads sit in a loop waiting to be woken up on a dedicated kernel mode wait event (stage 244). The system ensures that system calls for a user mode thread (or other actions affecting the kernel mode thread) always occur on the respective dedicated backing thread by shuffling the state to the backing thread before the system call is made (stage 246). Any modification or use of the backing thread data structure is

properly synchronized with the current caller thread (stage 248). The process ends at end point 250.

[021] Figure 4 illustrates one implementation of the more detailed stages involved in using dedicated backing threads for system calls for a user mode thread running on a kernel mode thread. In one form, the process of Figure 4 is at least partially implemented in the operating logic of computing device 100. The process begins at start point 270 with receiving notice that a particular user mode thread running on a kernel mode thread wants to make a system call (stage 272). If the user mode thread is currently running on its dedicated backing thread, then the backing thread executes the system call (stage 282). If, however, the user mode thread is not currently running on its dedicated backing thread (decision point 274), then the dedicated backing thread for the particular user mode thread is woken (stage 276), state is shuffled from the user mode thread to the backing thread (stage 278), and the kernel mode thread is put to sleep (stage 280). The backing thread then executes the system call (stage 282). In either case, after the backing thread executes the system call (stage 282), the kernel mode thread is then woken so it can regain control and load the context of the next user mode thread to be executed (stage 284). The process then ends at end point 286.

[022] Figure 5 illustrates one implementation of the stages involved in shuffling state from a user mode thread running on a kernel mode thread to a backing thread. In one form, the process of Figure 5 is at least partially implemented in the operating logic of computing device 100. The process begins at start point 290 with the system saving the register state of the current user mode thread running on the kernel mode thread (stage 292). The system puts the kernel

mode thread to sleep (e.g. by calling "signal and wait") (stage 294). The system wakes up the respective dedicated backing thread (stage 296) and restores the register state to the respective dedicated backing thread (stage 298). The process ends at end point 300.

[023] Figure 6 is a diagram 320 illustrating how a user mode thread running on a kernel mode thread is transitioned to a dedicated backing thread when a system call is made. The figure shows execution flow as a function of time for three user-schedulable threads. U_1-U_3 are the user mode threads, and $K_1 - K_3$ are the kernel mode threads. P is the "primary" kernel mode thread. The darker line shows the running user and kernel threads as a function of time. At T_1 the application calls a "fast switch to thread" routine and the runtime simply switches the user mode thread without changing which kernel mode thread is running. At T_2, the application makes a system call. Now the user mode thread is running on the wrong kernel mode thread so the system puts P_1 to sleep, wakes up K_2 (the backing thread for U_2), transfers context to it and runs the system call on K_2. At T_3, the system call returns.

[024] In one implementation, when the system call is completed, execution is moved back to the kernel mode thread. The backing thread waits so that if this or another kernel mode thread dispatches U_2, the backing thread will be ready to run system calls. In another implementation, which is an optimization that is shown in Figure 6, the user mode thread can continue to run on its backing thread. Then, at T_4, the application calls the "fast switch to thread" routine and execution is moved back to the kernel mode thread. The backing thread waits so

that if this or another kernel mode thread dispatches U_2, the backing thread will be ready to run system calls.

[025] Figure 7 illustrates one implementation of the stages involved in handling subsequent user mode executions. In one form, the process of Figure 7 is at least partially implemented in the operating logic of computing device 100. The process begins at start point 340 with determining whether it is the second or subsequent time that another user mode thread running on a kernel mode thread is selected for execution(e.g. of a system call or otherwise) (decision point 342). If not, then the process ends at end point 350. If this is the second or subsequent time, then the system determines if the execution is currently taking place on a backing thread (decision point 344). If execution is not currently taking place on a backing thread (decision point 344), then the current kernel thread is used for the subsequent execution of the user mode thread (stage 348). If however, the execution is currently taking place on a backing thread (decision point 344), then the backing thread and user mode thread are transitioned back to a base state (stage 346), the kernel mode thread is woken, and the user mode thread is run (stage 347). The process ends at end point 350.

[026] Figure 8 illustrates one implementation of the stages involved in providing thread affinity when using lazy kernel thread binding. In one form, the process of Figure 8 is at least partially implemented in the operating logic of computing device 100. The process begins at start point 370 with having the kernel mode thread wait on a wake up event so as not to preempt the backing thread whenever a kernel mode thread gives way to a dedicated backing thread (stage 372). Before the dedicated backing thread is woken, the thread affinity will

be set to the same processor core as the kernel mode thread to ensure instruction/cache locality is maintained (stage 374). When a backing thread is about to give way to a kernel mode thread, the kernel mode thread will be woken such that processing can continue (stage 376). The process ends at end point 380.

[027] Although the subject matter has been described in language specific to structural features and/or methodological acts, it is to be understood that the subject matter defined in the appended claims is not necessarily limited to the specific features or acts described above. Rather, the specific features and acts described above are disclosed as example forms of implementing the claims. All equivalents, changes, and modifications that come within the spirit of the implementations as described herein and/or by the following claims are desired to be protected.

[028] For example, a person of ordinary skill in the computer software art will recognize that the client and/or server arrangements, user interface screen content, and/or data layouts as described in the examples discussed herein could be organized differently on one or more computers to include fewer or additional options or features than as portrayed in the examples.

What is claimed is:

1. A computer-readable medium having computer-executable instructions for causing a computer to perform steps comprising:

decouple user mode and kernel mode portions of thread scheduling so that a particular user mode thread can be run on any one of a plurality of kernel mode threads (206).

2. The computer-readable medium of claim 1, wherein a user mode scheduler is responsible for dispatching of the particular user mode thread on a particular kernel mode thread of the plurality of kernel mode threads (210).

3. The computer-readable medium of claim 1, wherein when the particular user mode thread running on a particular kernel mode thread of the plurality of kernel mode threads wants to perform an action that could affect the particular kernel mode thread, a state shuffling process is performed to shuffle state from the particular user mode thread to a respective dedicated backing thread (208).

4. The computer-readable medium of claim 3, wherein the action is a system call (208).

5. The computer-readable medium of claim 3, wherein the state shuffling process is operable to save a register state of the particular user mode thread (292).

6. The computer-readable medium of claim 5, wherein the state shuffling process is further operable to restore the register state to the respective dedicated backing thread (298).

7. The computer-readable medium of claim 6, wherein the state shuffling process is further operable to put the particular kernel mode thread to sleep (294).

8. The computer-readable medium of claim 7, wherein the state shuffling process is further operable to wake up the respective dedicated backing thread (296).

9. A method for using a dedicated backing thread for a system call for a user mode thread running on a kernel mode thread comprising the steps of:

receiving notice that a particular user mode thread running on a particular kernel mode thread wants to make a system call; (272)

waking a dedicated backing thread that has been assigned to the particular user mode thread (276);

shuffling state from the user mode thread to the dedicated backing thread (278);

putting the particular kernel mode thread to sleep (280); and

executing the system call using the dedicated backing thread (282).

10. The method of claim 9, wherein the waking, shuffling, and putting stages are only performed if the user-mode thread is not already running on the dedicated backing thread (274).

11. The method of claim 9, further comprising:

waking the particular kernel mode thread so the particular kernel mode thread can regain control (284).

12. The method of claim 9, wherein the particular kernel mode thread will remain asleep until receiving a waking event so as not to preempt the backing thread (372).

13. The method of claim 9, wherein before the backing thread is woken, setting a thread affinity to a same processor core as the particular kernel mode thread (374).

14. The method of claim 13, wherein the thread affinity is set to the same processor to ensure that instruction and cache locality is maintained (374).

15. The method of claim 9, wherein on a subsequent time that a subsequent user mode thread is selected for execution, and execution is currently taking place on a particular corresponding backing thread (342), the particular corresponding backing thread and subsequent user mode thread are transitioned to a base state (346), the particular kernel mode thread is woken, and the subsequent user mode thread is run (347).

16. A computer-readable medium having computer-executable instructions for causing a computer to perform the steps recited in claim 9 (200).

17. A method for shuffling state from a user mode thread running on a kernel mode thread to a backing thread comprising the steps of:

receiving notice that a particular user mode thread running on a particular kernel mode thread wants to make a system call (272);

saving a register state of the particular user mode thread (292);

putting the particular kernel mode thread to sleep (294);

waking up a respective backing thread that was assigned to the particular user-mode thread (296); and

restoring the register state to the respective backing thread (298).

18. The method of claim 17, further comprising:

executing the system call using the respective backing thread (282).

19. The method of claim 18, further comprising:

waking up the particular kernel mode thread so it can regain control (284).

20. A computer-readable medium having computer-executable instructions for causing a computer to perform the steps recited in claim 17 (200).

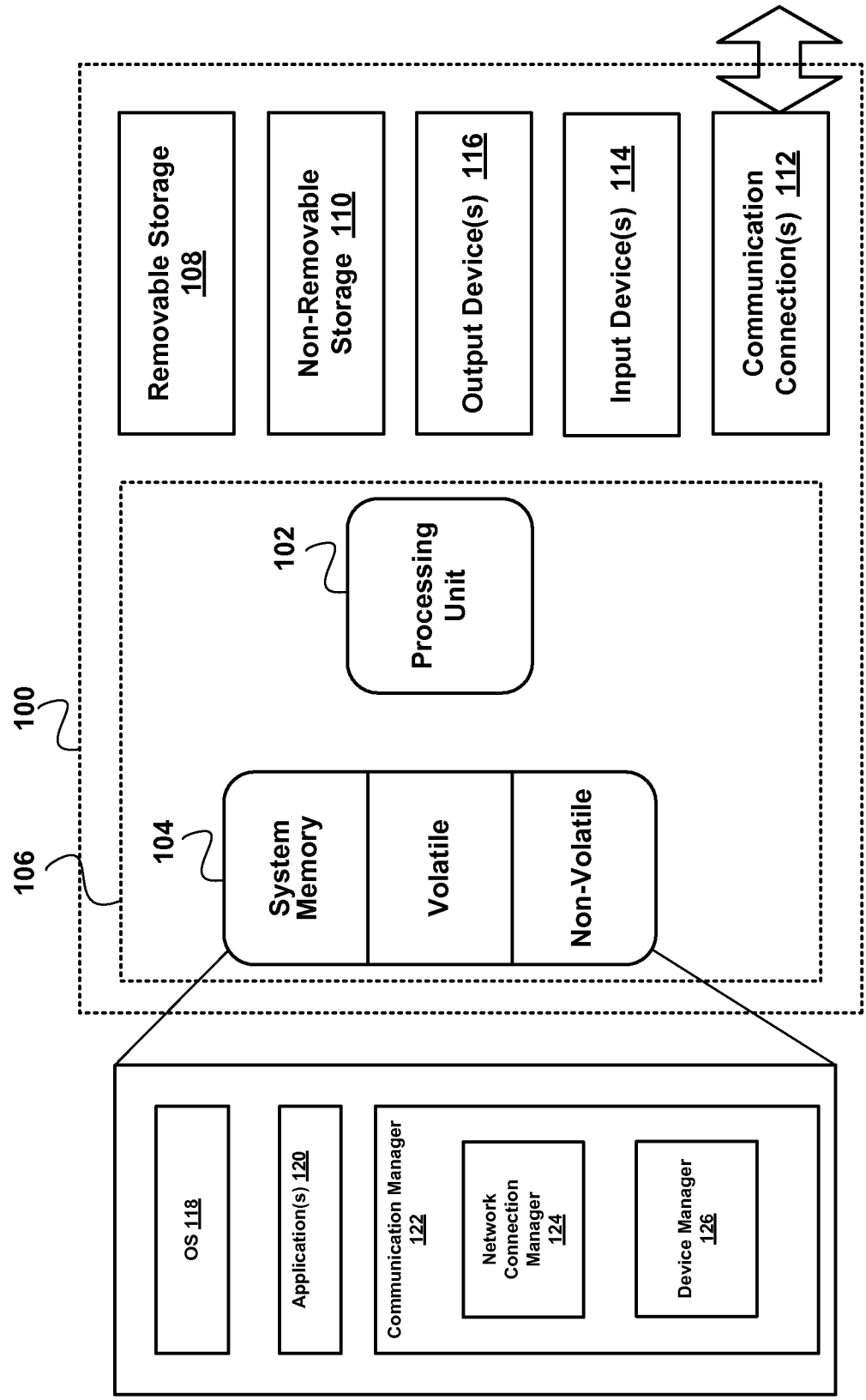
Removable Storage 108
Non-Removable Storage 110
Output Device(s) 116
Input Device(s) 114
Communication Connection(s) 112
102
Processing Unit
100
106
104
System Memory
Volatile
Non-Volatile
OS 118
Application(s) 120
Communication Manager 122
Network Connection Manager 124
Device Manager 126

Fig. 1

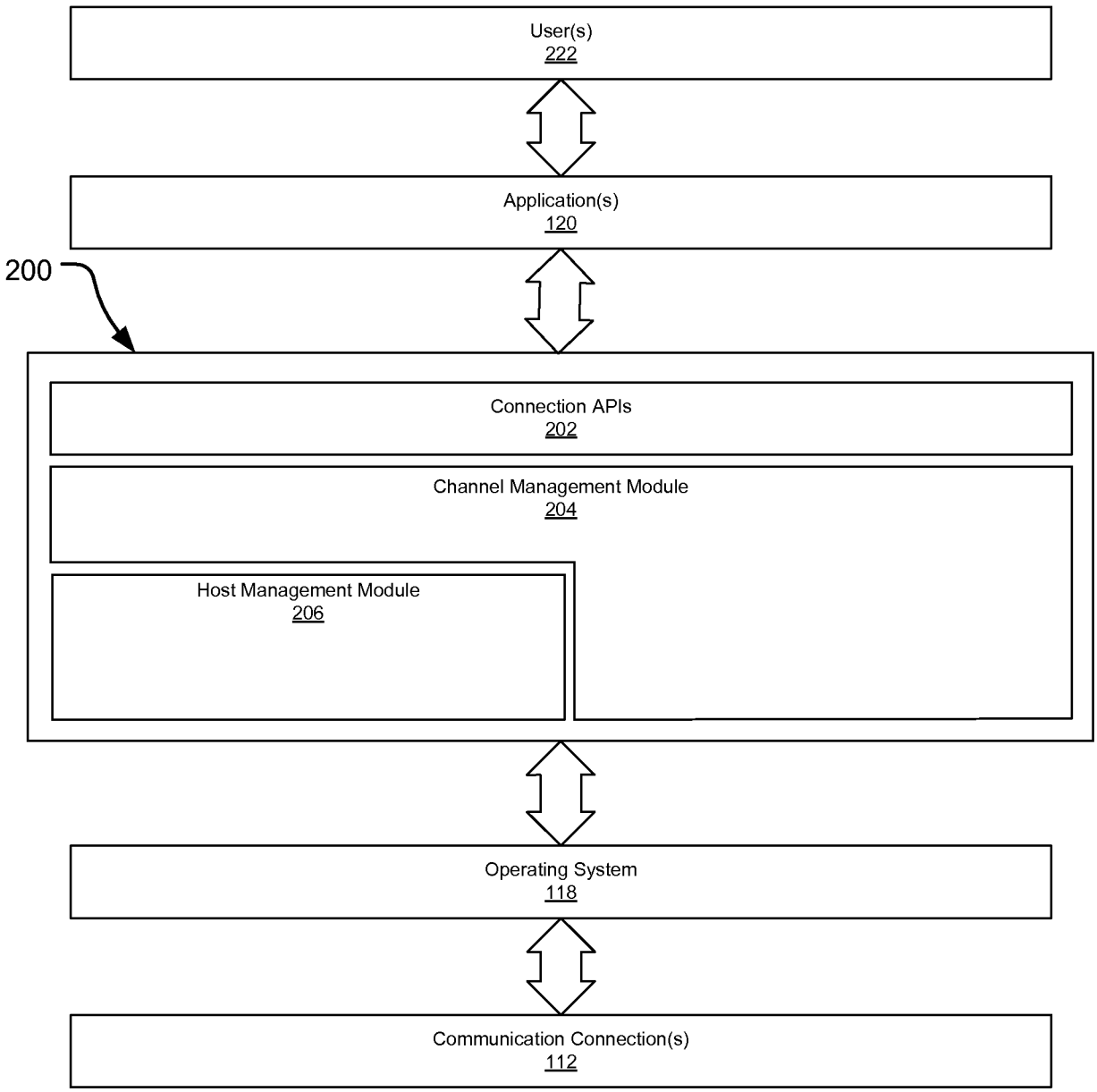
User(s)
222
Application(s)
120
200
Connection APIs
202
Channel Management Module
204
Host Management Module
206
Operating System
118
Communication Connection(s)
112

Fig. 2

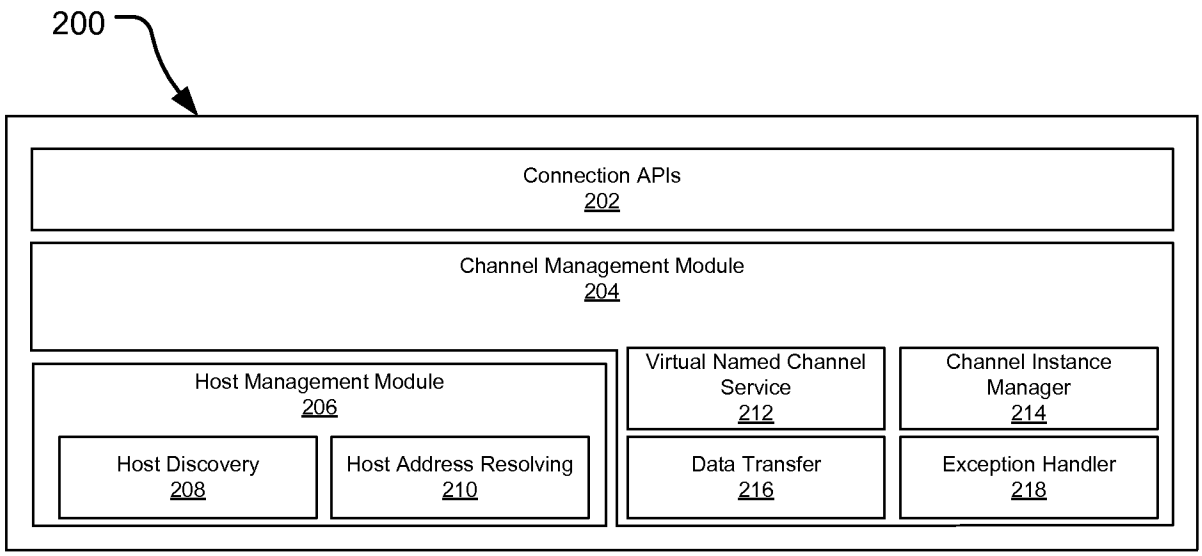
200
Connection APIs
202
Channel Management Module
204
Host Management Module
206
Host Discovery
208
Host Address Resolving
210
Virtual Named Channel Service
212
Channel Instance Manager
214
Data Transfer
216
Exception Handler
218

Fig. 3

400

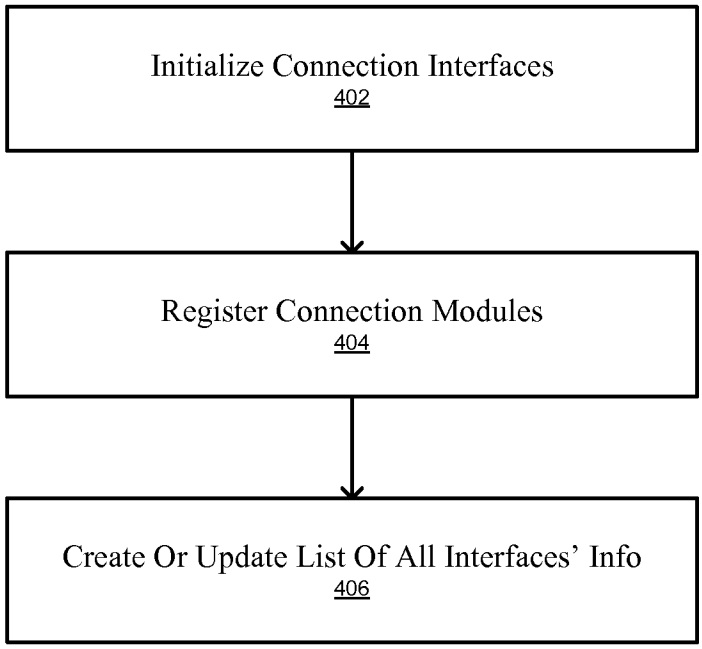
Initialize Connection Interfaces
402
Register Connection Modules
404
Create Or Update List Of All Interfaces' Info
406

Fig. 4

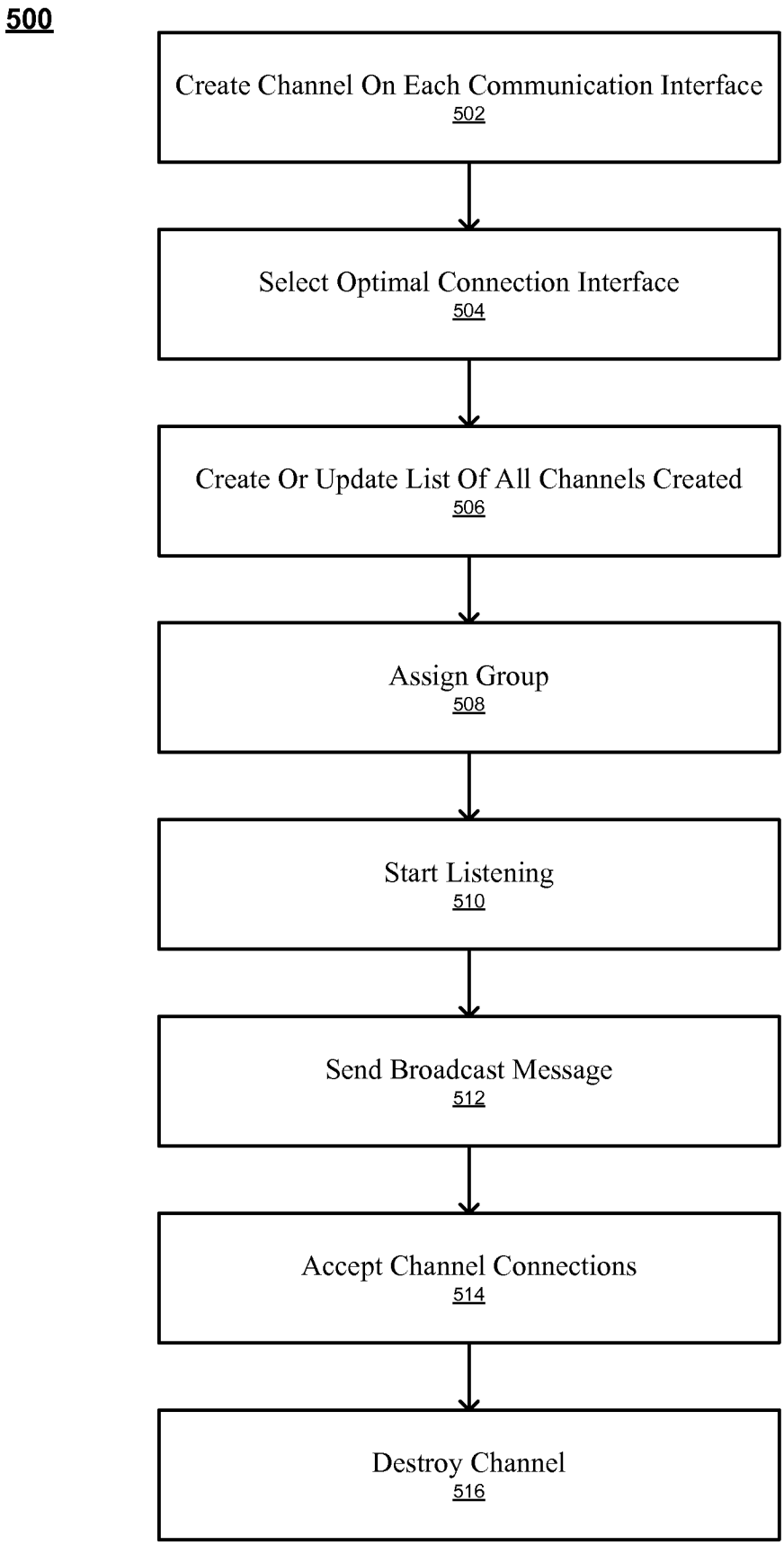
500
Create Channel On Each Communication Interface
502
Select Optimal Connection Interface
504
Create Or Update List Of All Channels Created
506
Assign Group
508
Start Listening
510
Send Broadcast Message
512
Accept Channel Connections
514
Destroy Channel
516

Fig. 5

600

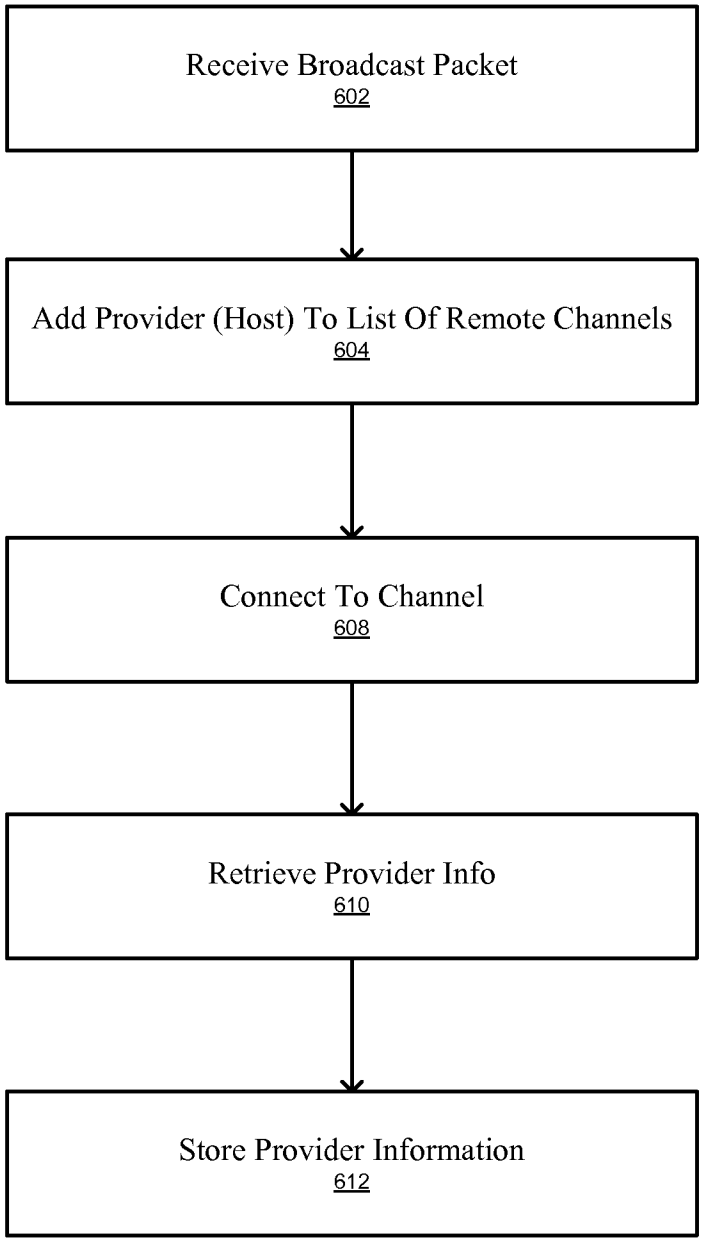
Receive Broadcast Packet
602
Add Provider (Host) To List Of Remote Channels
604
Connect To Channel
608
Retrieve Provider Info
610
Store Provider Information
612

Fig. 6

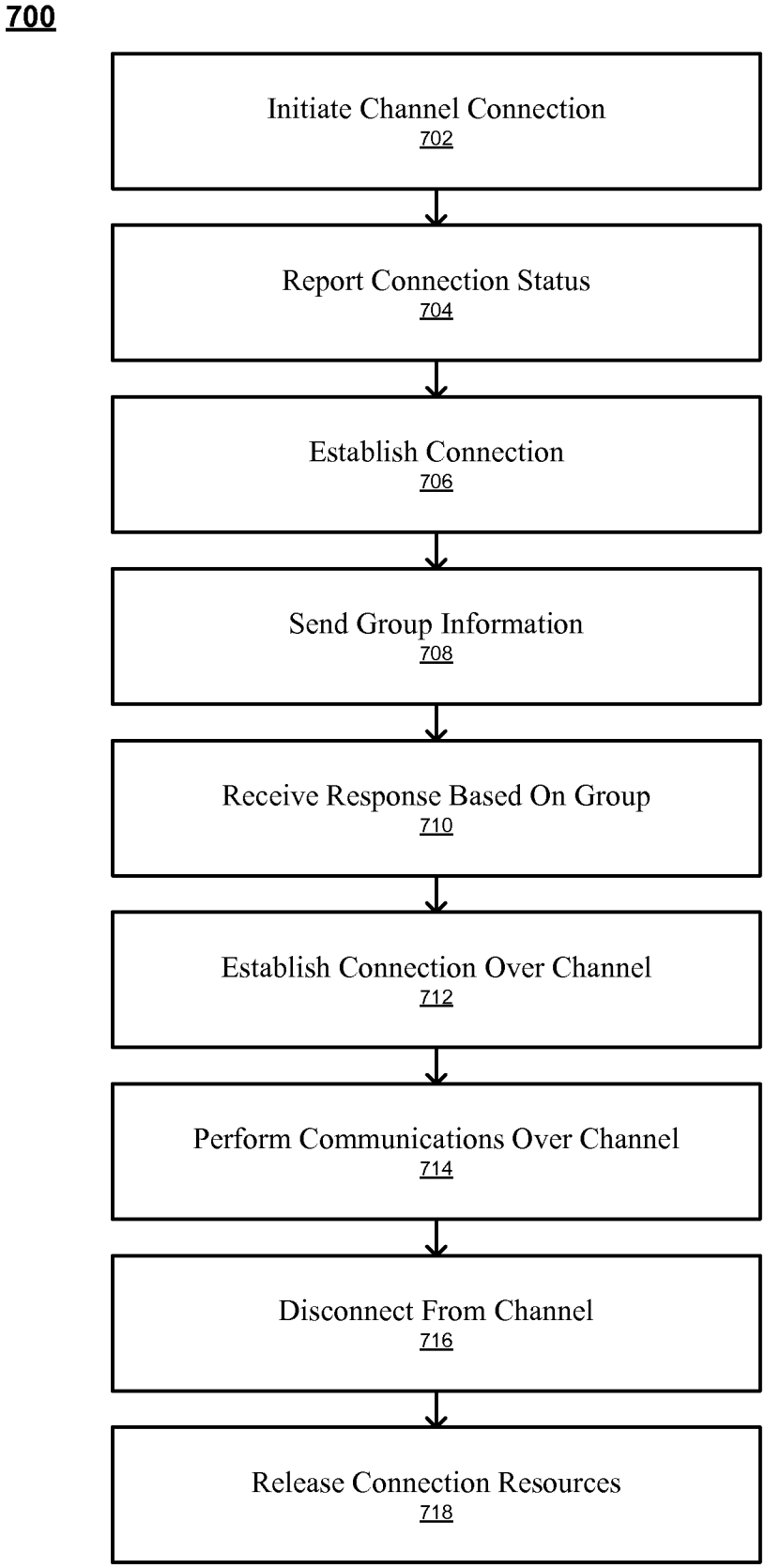
700
Initiate Channel Connection
702
Report Connection Status
704
Establish Connection
706
Send Group Information
708
Receive Response Based On Group
710
Establish Connection Over Channel
712
Perform Communications Over Channel
714
Disconnect From Channel
716
Release Connection Resources
718

Fig. 7

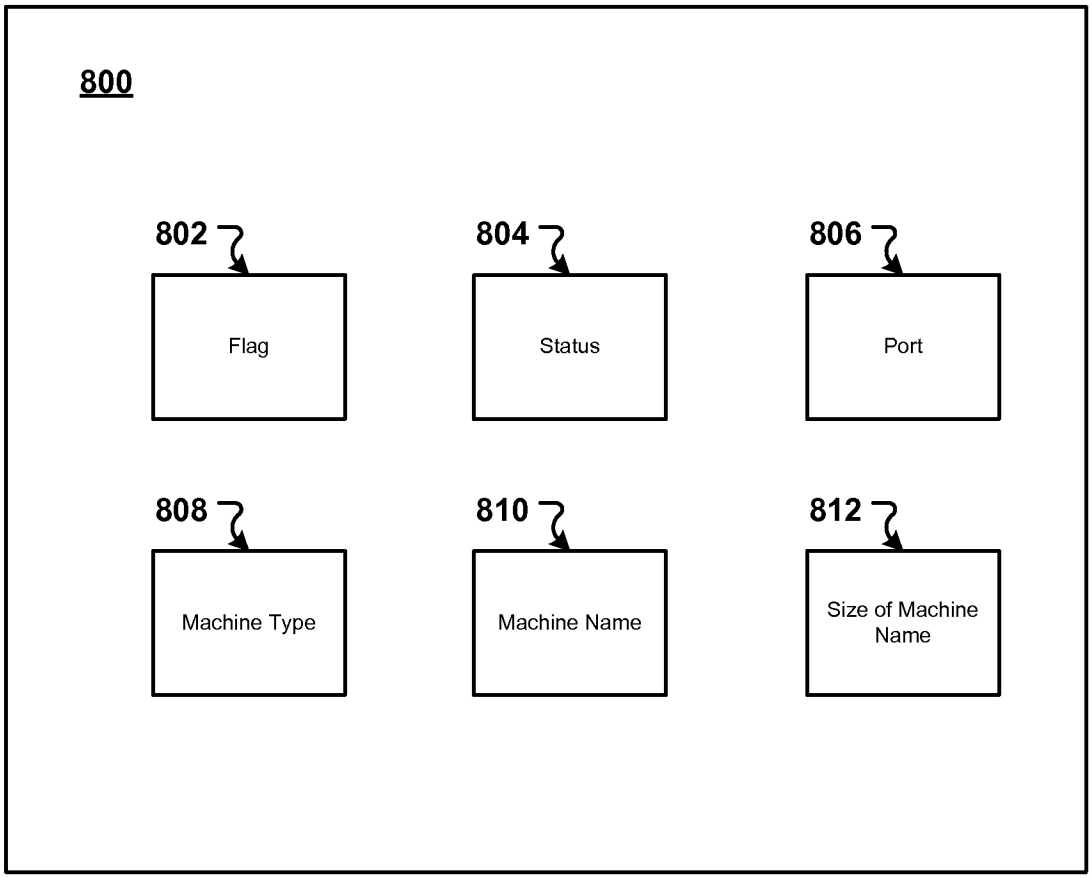
800
802
Flag
804
Status
806
Port
808
Machine Type
810
Machine Name
812
Size of Machine Name

Fig. 8

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2008/064812

A. CLASSIFICATION OF SUBJECT MATTER

G06F 9/46(2006.01)i

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)

IPC 8 G06F

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

Korean Utility models and applications for utility models since 1975
Japanese Utility models and applications for utility models since 1975

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

eKIPASS(KIPO internal) "thread", "scheduling", "user mode", "kernel mode"

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X A	SILBERSCHATZ, A. et al. Operating System Concepts. 7th ed. Wiley & Sons, Inc. 2004, ISBN 0-471-69466-5, pp. 133-159, 167-186.	1, 2 3-20
A	US 6 349 355 B1 (DRAVES, P. D. et al.) 19 February 2002 See abstract, column 3, lines 35-38, column 4, lines 51-53.	1-20
A	Woodside, M. C. et al. The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. IEEE Transactions on Computers. January 1995, Vol. 44, No. 1 pages 20-34, ISSN: 0018-9340. See abstract, figures 1, 2, 4-7 and their descriptions.	1-20

☐ 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 application or patent but 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 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 to involve an inventive step when the document is combined with one or more other such documents,such combination being obvious to a person skilled in the art
"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
29 AUGUST 2008 (29.08.2008)	**29 AUGUST 2008 (29.08.2008)**

Name and mailing address of the ISA/KR	Authorized officer
Korean Intellectual Property Office Government Complex-Daejeon, 139 Seonsa-ro, Seo-gu, Daejeon 302-701, Republic of Korea	LEE, Sang Hun
Facsimile No. 82-42-472-7140	Telephone No. 82-42-481-5914

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US2008/064812

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 6 349 355 B1	19.02.2002	None	