159159

COMPONENT-BASED EMBEDDED SYSTEM AND METHOD THEREFOR

Field of the Invention

The present invention relates to a component-based embedded system and more particularly to a component-based embedded system that supports an extensible service interface (ESI).

Background of the Invention

The combination of optimized hardware platform and function-specific software applications to provide a custom-made device is commonly referred to as an embedded system, and information appliances are an example of products that are built on embedded systems.

Advantageous lower cost and ease of use of custom-made devices provide the motivation for producing such devices. However, in general each custom-made device is necessarily independently developed, and there is limited re-use of software between different types of custom-made devices that support different functionalities. For example, a custom-made information appliance for storage of data on a compact disk would not share many common components with a custom-made information appliance for accessing the Internet.

FIG. 1 shows a known embedded system of a custom-made information appliance 100, which has components in user space 105, kernel space 110, and hardware space 115. User space 105 and kernel space 110 are embodied in software, while the hardware space 115 represents the actual hardware of the appliance 100. The user space 105 includes, for example, one application 120 and a corresponding application library 125, that together support particular functionalities of the appliance 100; and a system library 130 allows the application 120 and the corresponding application library 125 to communicate

G00002

with components in the kernel space 110. The kernel space 110 comprises a kernel 135 with a virtual memory system 140, a signal module 145, a virtual file system (VFS) 150, and, for example, one device driver 155. In addition, the hardware space 115 includes a hardware device 160, which is uniquely associated with the device driver 155. For example, in an Internet appliance, the hardware device can comprise a network interface card and the device driver would then comprise a corresponding device driver for the network interface. Alternatively, in a storage appliance, the hardware device can comprise a hard disk drive, and the device driver would then comprise a corresponding device driver for the hard disk drive. Typically, the device drivers are produced and provided by the hardware device manufacturers.

The VFS 150 abstracts the interface to the device driver 155 as a device file (not shown), which has a unique identifier allowing the kernel 135 to determine the associated device driver 155 of the device 160. In operation, communication from the application 120 to the device 160 is sent 170A from the application 120 to the system library 130, which then sends 170B an associated communication to the VFS 150. The VFS 150 then passes 170C data that is read and written from the device file to the device driver 155, which then acts on the data. The data includes requests, such as control data, or ordinary data to be passed to the device 160. An example is sending a data packet to a network interface card for transmission.

In response, the device driver 155 returns 170D data or status information to the VFS 150, which sends 170E an associated response to the system library 130, which in turn sends a corresponding response to the application 120. The data transfer between a device driver 155 and the application 120 takes place through a copy process, whereby the data that is to be passed from one to the other is copied from the user space 105 to the kernel space 110 or vice versa.

The application library 125 provides two types of functions, namely a utility function and a system call function. The utility function performs action completely in user space 105, and an example of a utility function is performing calculations.

The system call function is similar to the process described earlier except that now instead of communicating with the system library 130 directly, the application 120 makes a function call to the application library 125, which then makes a system call via the system library 130, and then provides a response to the application 120.

Whether the application 120 communicates directly with the system library 130 or via the application library 125, in either mode, the application 120 must know the particular function call that it needs to make to the system library 130 or to the application library 125. In addition, the application 120 must also know the respective parameters that need to be provided with the particular function call.

The signal module 145 provides a variety of signals to the application 120. The application 120 has the ability to respond to some of the signals, however, for some other signals, the application 120 may be forced by the kernel to stop. To illustrate, by way of an example, where the application library 125 supports memory allocation; when the application 120 sends 175A a function call to the application library 125 to release a specified amount of memory space, the application library 125 sends 175B a communication to the system library 130. The system library 130, in turn sends 175C a system call to the virtual memory system 140, which manages the memory. Then, when a bug in the application library 125 causes more then the specified memory that was allocated by the operating system to be freed by the application 120, via the application library 125, the virtual memory system 140 communicates 175D with the signal module 145. Consequently, the signal module 145 sends 175E a signal to the

application 120, which causes the application 120 to terminate abruptly or crash.

It should be noted that there are variety of signals that are provided by the signal module 145, and that applications, such as the application 120, can respond to some signals; but for other signals, as in the example above, the signal causes an application to crash. Also, from the example, it will be appreciated that although the bug was in the application library 125, the operating system causes the application 120 to crash. This is because the operating system knows that the function call was sent 175A by the application 120, and that the application library 125 is a part of the memory allocated to the application 120.

Libraries, such as the application library 120 are used extensively in conventional application development to provide an abstraction layer to system calls, and also to provide a set of commonly used functions, which simplify application development. There are static and dynamic libraries, and an application must be linked to a particular library in order to use the particular library. When an application is linked to a static library, a copy of the library functions is made in the application executable program. When an application is linked to a dynamic library, the application is statically linked to a dynamic loader, which loads other libraries as required. The code of the dynamic library is not exclusive to the application, and is not copied into the application executable program. Whether static or dynamic libraries are employed, any changes to the library affect the link with the application, and therefore requires the application, such as the application 120, to be recompiled.

Hence, a disadvantage of the conventional embedded system is the need to recompile an application whenever changes are made to a corresponding application library.

The link or interface between an application and an application library is referred to as an application-programming interface (API).

The API defines functions and parameters, and changes in the API can result in linking errors. To mediate the occurrence of linking errors, application libraries are versioned, and this allows compatibility to be maintained across major versions. Consequently, in order for an application to take advantage of new functionality in an application library, the application will need to be re-written to support the changes in the API.

Thus, another disadvantage of a conventional embedded system is the need to re-write an application to add any newly developed functionality to an application library.

Using an information appliance that supports a particular function as an example, the conventional approach for developing a custom-made device involves combining an optimized hardware platform and function-specific software applications.

First, a particular optimized hardware platform is created with the necessary hardware components to provide particular functionalities. Next, particular function-specific software applications are either written, or adapted from existing software applications, to operate on the particular optimized hardware platform. Typically, the software is developed using what is sometimes referred to as the monolithic application development model. In accordance with this model, a software developer would need to write much of the function-specific software applications.

Hence, a disadvantage of the conventional method of monolithic application development for developing the particular function-specific software applications, is that it is not component-based. In addition, monolithic application development does not readily support re-use, and is inflexible to adaptation for operation when the application, operating system and/or the hardware is changed.

Known models of component-based software programming provide an improvement to the monolithic application development model, and one such model of component-based software

programming is common object request broker architecture (CORBA), which provides a source code interface. The CORBA model provides a common interface written in an interface definition language (IDL), which supports communications between different CORBA processes. For example, a program in the C programming language, that is to be incorporated as a CORBA process, must first be written using the IDL and then compiled using an IDL compiler to produce a C language program source file. In addition to compiling the source file, the C language source program must also link to a C language CORBA library. The CORBA library allows two or more different CORBA processes to communicate across different computer systems, even when the processes are in different programming languages.

A disadvantage of the CORBA model is that it does not provide run time flexibility, as any changes to either the C language source program or the C language CORBA library, requires re-compilation.

Brief Summary of the Invention

The present invention seeks to provide a component-based embedded system, and method therefor, which overcomes or at least reduces the abovementioned problems of the prior art.

Accordingly, in one aspect, the present invention provides an embedded system comprising:

at least one service provider for registering to provide at least one service, the at least one service provider for receiving a service request associated with the at least one service, for having the at least one service performed, and for providing a performance status indicating the performance of the at least one service;

at least one service consumer for communicating one of a plurality of service requests for the at least one service, and for receiving the performance status; and

at least one service manager being operably coupled to the at least one service provider and to the at least one service consumer, the at least one service manager for receiving the one of the plurality of service requests from the at least one service consumer and for providing the service request to the at least one service provider, and the service manager for receiving the performance status from the service provider, and the service manager for sending the performance status to the at least one service consumer.

In another aspect the present invention provides a method for an embedded system comprising the steps of:

a) providing at least one service provider, at least one service consumer, and at least one service manager being operably coupled to the at least one service provider and to the at least one service consumer;

b) the at least one service consumer communicating a service request for a particular service to the at least one service manager;

c) the at least one service manager communicating the service request for the particular service to the at least one service provider;

d) the at least one service provider communicating a notification to the at least one service manager; and

e) the at least one service manager communicating the notification to the at least one service consumer.

Brief Description of the Drawings

An embodiment of the present invention will now be more fully described, by way of example, with reference to the drawings of which:

FIG. 1 shows a functional block diagram of an embedded system in accordance with the prior art;

FIG. 2 shows a functional block diagram of an embedded system in accordance with the present invention;

FIG. 3 shows a simplified functional block diagram of the embedded system in FIG. 2;

FIG. 4 shows a flowchart detailing the operation of the embedded system in FIG. 3;

FIG. 5A and 5B show operation of the communications sub-system in FIG. 2;

FIG. 6A and 6B show operation of the event sub-system in FIG. 2;

FIG. 7 shows a flowchart detailing the operation of the service processor in FIG. 2;

FIG. 8 shows a schematic of data structures in the service memory in FIG. 2; and

FIG. 9 shows a message format employed in the embedded system in FIG. 2.

Detail Description of the Drawings

A component-based embedded system is disclosed which provides a services model comprising service consumer components that use services; service provider components that provide the required services, and a service manager which performs the role of a central service authority that regulates the provision of the services between the service consumer components and service provider components. An extensible service interface (ESI) is defined, which enables communication between the service consumers and the service manager, and enables communication between the service providers and the service manager. The ESI thus provides an abstraction layer between the service consumers and the service provider, and the service manager, where the service consumers and service providers do not need to communicate directly with each other. Instead, the service consumers and service providers need only communicate with the service manager via the common ESI.

The abstraction provided by the services model and the ESI advantageously allows component based programming that extends the concept of modularity to the object code level, in contrast to the source code level of the prior, as taught by CORBA. Components of the services model can now be modified at run time, allowing a change in functionality of the components to be effected without the need for re-compilation or re-integration. In addition, as the components share a common ESI, components can be re-used. For example, where an application program, which represents a service consumer, was developed for a specific hardware and/or operating system, which forms a particular service provider, the application program can be advantageously re-used on a different hardware and/or operating system, which forms a different service provider.

With reference to FIG. 2 a component-based embedded system 200, in accordance with the present invention, has components in the user space 105, the kernel space 110, and the hardware space 115. The user space 105 and the kernel space 110 are embodied in software, and the hardware space 115 represents the actual hardware of the system 200. The component-based embedded system 200 comprises a service consumer 205, a service manager 210, a service provider 215, and a service consumer-provider 217, each shown delineated within broken line boundaries.

The service consumer 205 operates in user space 105, and can comprise an application program 225 that communicates via the system library 260 with components in kernel space 110. The service provider 215 includes a service provider manager 245, which operates in user space 105, a corresponding service provider driver 250, which operates in kernel space 110, and a corresponding device 255, in hardware space 115. The service provider manager 245 is coupled to communicate with the service provider driver 250 via the system library 260, and the service provider driver 250 in turn is coupled to communicate with the device 255, which performs a particular service.

The service manager 210 includes a configuration manager 265, that operates in user space 105; and a service memory 270, a service processor 275, a communication sub-system 280, and an event sub-system 285, all four of which operate in kernel space 110.

The service provider-consumer 217 comprises an application that includes an application library 235, which has been adapted to operate in an ESI shell 240. The ESI shell 240 enables the application library 235 to communicate via the system library 260 with components in the kernel space 110, in accordance with the ESI. The service provider-consumer 217 acts as a service provider to the application 225, where the service is provided through a function call. In addition, the service provider-consumer 217 also acts as a service consumer to other components.

The communication sub-system 280 is coupled to facilitate communication with the application 225 via the system library 260, and coupled to facilitate communication with the application library 235, via the ESI shell 240 in accordance with an ESI protocol, and the system library 260. In addition, the communication sub-system 280 is also coupled to facilitate communication with the service provider manager 245 and the configuration manager 265, in accordance with the ESI protocol, via the system library 260. Both the service provider manager 245 and the configuration manager 265 are coupled to communicate with the service memory 270, and the service processor 275, via the system library 260. The service processor 275 is coupled to communicate with both the communication sub-system 280 and the service memory 270, and is coupled to provide information to the event sub-system 285.

With additional reference to FIGS. 3 and 4 the operation 400 of the component-based embedded system 200 starts 405 with the service consumer 205 determining 410 whether it needs to request a particular service. When it does, the service consumer 205 sends 415 a service request 305 for the particular service to the service manager

210. Upon receipt of the service request, the service manager 210 determines 420 whether the service provider 215 is available to provide the requested service, assuming for illustration purposes that the service provider 215 provides the requested service.

When the service manager 210 determines that the service provider 215 is available to provide the requested service, the service manager 210 sends 425 a corresponding service request 310 to the service provider 215. In response, the service provider 215 performs 430 the requested service, and provides 435 a response to the service manager 210 indicating performance status 315 of the service request. The service manager 210 subsequently receives the performance status, and provides 440 a corresponding performance status 320 to the service consumer 440, after which the operation 400 returns to step 410 and continues as described above.

Messages are used for communication between the service consumer 205, the service manager 210, and the service provider 215. The service manager 210 advantageously routes messages in tagged or plain text format from one component, such as the service consumer 205, to another component, such as the service manager 215, in a peer-to-peer or a client-server component–based architecture. This allows the service consumer 205 to advantageously access system or service provider 215 services, without knowing how to, or being able to, communicate directly, with the system or the service provider 215.

The messages are sent in accordance with the ESI protocol to the service processor 275, which consists of a kernel module with a device file as its interface, which will be referred to hereinafter as a kernel device file interface. The service processor 275 also creates a file (not shown) in the proc file system (not shown) of the kernel to report status information. The kernel module can be dynamically loaded in order to initialize the service processor 275, and the kernel module registers its internal functions to be called when file operations are performed on the kernel device file interface. The service processor

275 provides "replacements" for the default VFS operations such as read, write, ioctl, mmap, munmap etc., as such operations or function calls will be known to one skilled in the art. The "replacements" comprise predefined functions for communicating with applications, such as the application 225. This makes it possible to trap file operations on the kernel device file interface and treat the data written to the kernel device file interface within the kernel module as an ESI message.

Through the read, write and mmap file operations, the service processor 275 allows the application 225, the application library 235 in the ESI shell 240, the configuration manager 265, and the service provider manager 245, all of which are treated as application programs by the service processor 275, to have access to the communication sub-system 280 and the event sub-system 285. In addition, the service processor 275, also allocates memory in kernel space 110 in the service memory 270, to store a running configuration of the embedded system 200, and makes the running configuration available to the applications 225, 217, 265 and 245 in user-space 105 through a process called mmaping. The mmaping process makes kernel memory available to a user-space process by manipulating the user-visible virtual address space. The allocated memory is called ESI shared memory since any of the ESI components are allowed access.

Hence, the present invention provides a services view of an embedded system, comprising service consumer components 205, service provider components 215, and a service manager component 210, which are coupled and communicate in accordance with the ESI. A service consumer component, such as the application 225, that wishes to request a service, uses the communication sub-system 280 to send a message to the service processor 275. A service provider component 245 that wishes to provide a service registers its intent with the event sub-system 285. The service processor 275 uses the event sub-system 285 to notify the service provider component 245 of

a service request. The service provider component 245 can then have the combination of the service provider driver 250 and the device 255, perform the required action, and send a reply to the requesting service consumer component 225 using the event sub-system 285 or communication subsystem 280. The ESI protocol is used by the service provider component 245 when registering its service, and the ESI protocol is also used by the service consumer component 205 that is looking for available services. In addition, the ESI protocol is also used by the service provider 215 when sending a service performance status, such as a success or failure indicator, to the service manager 210, as well as by the service manager 210, when reporting an error back to the service consumer 205.

Data communicated by the communications sub-system 280 comprise messages, where data is transferred between the applications 225, 217, 265 and 245 and the service processor 275. Alternatively, the messages can comprise inter-process communication (IPC), where data is transferred between the applications 225, 217, 265 and 245.

In addition, the communication sub-system 280 supports two mechanisms for communication. With reference to FIG. 5A, the first mechanism uses shared memory space 505 in the service memory 270, where the applications 225, 217, 265 and 245 read and write to the shared memory space 505. A transmitting application writes data to the shared memory space 505, and the receiving application(s) read the data from the shared memory space 505. Consequently, there is no need to copy the data that is communicated, and this is referred to as a zero copy operation.

With reference to FIG. 5B, the other mechanism for communication involves using a device file interface 510 to the service processor 275. The data 515 to be communicated is written to the device file interface 510 by a transmitting application; and the service processor 275 stores the data 515 in a data buffer of the service

processor 275. Copies 520 and 525 of the data are transmitted 530 to the receiving applications.

The two data communication options allow an application written in a low-level programming language to make more efficient use of the embedded system 200, while maintaining compatibility for higher level programming languages. An application in a low level programming language can directly access the service memory 270 through the mmap operation. This allows zero-copy access to the data, which saves time and memory, while taking action on a service request. An application, such as the application 225, in a high level programming language, which may not have direct access to the service memory 270 can still use messages to communicate with the service processor 275 to request or provide services. However, some types of services can only be provided by an application written in a low level language, and it is generally the case that since most service providers 215 use such low level languages, an improved response time to the service consumer 205 is achieved.

When using either of the two communication mechanisms, a defined message format is employed. The message content typically consists of an XPath expression and a message type. Messages are used for all communication in the embedded system 200 including reading and writing to the running configuration stored in the service memory 270, for event registration and delivery, and for IPC.

The event sub-system 285 allows the applications 225, 217, 265 and 245 to be notified by the service processor 275 upon the occurrence of a predetermined event. Several different event notification mechanisms are supported to allow different applications with different capabilities to have access to these event notification mechanisms. Events form the basis of all actions in the embedded system 200. Various actions, including provision of services between the service provider 215 and the service consumer 205, error handling,

as well as IPC, are handled using the event notification mechanisms provided by the event sub-system 285.

The event sub-system 285 supports two forms of events. A first form involves providing explicit notification to the applications 225, 217, 265 and 245. The other form involves polling by the applications 225, 217, 265 and 245 to check the status of a condition. The applications 225, 217, 265 and 245 must register, in advance, for events using a registration message, and the applications 225, 217, 265 and 245 must also specify the type of notification it expects when it registers.

With reference to FIG. 6A, explicit notification is where a first application registers 605 for explicit notification in the running configuration 610 that is stored in the service memory 270. When the event for the notification is triggered 615, say by another application, the event sub-system 285 notifies 620 the first application.

With reference to FIG. 6B, polling is where an application registers 630 in the running configuration 610, and subsequently, the application polls 635 the running configuration 610 for event notification.

The event sub-system 285 uses a transaction model, where each action taken by an application is assigned a transaction ID, which is encoded into the header of a message. The components of the embedded system 200 use the transaction ID to track particular transactions. For example, a notification by the service processor 275 about a service request to the service provider manager 245 includes a transaction ID. When an error occurs while taking the action required by the service request, the service provider manager 245 sends back a message to the service processor 275 containing the transaction ID and a respective error message. In this way, the service processor 275 will know how to route the error message.

The service processor 275 must be able to route a service request from a service consumer 205 to the correct service provider 215.

Consequently, the service processor 275 maintains a mapping between certain parameters that can be set by the service consumer 205 i.e. the applications 225, 217, 265 and 245, and the service providers, such as the service provider 215, that actually effects a change in the system. The service processor 275 does this using the running configuration 610, which is an extensible markup language (XML) based database stored in the service memory 270. The running configuration 610 stores the current state of the embedded system 200 in terms of the current values of various system and application parameters.

An example of a system parameter would be the IP address of the system, whereas an application parameter would be a configuration variable used by the applications 225, 217, 265 and 245. The service provider 215 uses a registration mechanism to assume responsibility for changes in certain areas of the running configuration 610. The service processor 275 notifies the service provider 215 of any change in the running configuration 610, which may be caused by an application 225 or other service providers, at which point the service provider 215 decides whether this change requires it to perform a service. By extension, the database stored in persistent storage as an XML document is called the stored configuration. The running configuration 610 is generated in kernel space 110 by an application called the configuration manager 265, which parses the stored configuration using the simple API for XML (SAX) parser, which is an event based XML parser, and generates a tree structure.

Any information in the running configuration 610 can be accessed using XPath expressions. XPath is software language that is used for addressing parts of an XML document. Since the original XPath specification is defined to work with document object model (DOM) tree, the configuration manager 265 provides its own internal evaluation algorithms to efficiently search the data structure of the running configuration 610. A DOM is a tree representation of an XML

document, and the XPath implementation, as employed herein, is a subset of the original world wide web (WWW) specification.

The running configuration 610 works together with the event sub-system 285 to provide a service interface. An application, such as the application 225, that wishes to request a service modifies some data in the running configuration 610 by addressing it using a predefined XPath expression. The service provider manager 245, of the service provider 215 that provides the requested service, will have registered selecting the event sub-system 285 mechanism to be used for notification when that data is changed. The service provider 215 can then perform the service.

The configuration manager 265 manages the running configuration 610 from user space 105, by reading the running configuration 610 from stored XML files, and creating the running configuration 610 accordingly. The configuration manager 265 parses XPath expressions, and makes modifications to the running configuration 610. The configuration manager 265 is another application, which comprises another component of the embedded system 200, and communicates in accordance with the ESI protocol. The event sub-system 285 is used by the service processor 275 to provide a notification to the configuration manager 265, when the service processor 275 receives an XPath expression.

The service processor 275 implements the shared memory in the service memory 270 by allocation of memory space using the __get_free_pages function of the kernel of the embedded system 200. The service processor 275 implements an ioctl call, which allows the configuration manager 265 to allocate an appropriate amount of memory for the running configuration 610.

The mmap operation is performed by the service processor 275, redefining the default mmap definition in the file_operations structure of the kernel. The replacement mmap function replaces the known vm_ops struct in the process's appropriate vm_area_struct struct. This

allows the default nopage function call to be intercepted to return the page allocated by the earlier __get_free_pages function.

With reference to FIG. 7, the process 700 of mmaping in the service memory 270 starts 705, for example, with the service provider manager 245 performing 710 a mmap operation to access the running configuration 610 in the service memory 270. The kernel then adds 715 the virtual address of the running configuration 610 into process address space. The application 225 then attempts 720 to access the running configuration 610 in the service memory 270, causing a page fault to occur 725. Subsequently, the kernel calls 730 the no-page function of the service processor 275. The service processor 275 then maps 735 the virtual address, addressed by the service provider manager 245, into the page address of the appropriate physical page in random access memory of the embedded system 200, after which the kernel adds 740 the page address into the process address space of the service provider manager 245. The service provider manager 245 is also referred to as the client that performs the mmapping operation. The application 225 can now read/write 745 to the running configuration 610, and the process 700 ends 750.

In a similar way as that described above, the service processor 275 allocates and makes mmap-able several other memory areas of the service memory 270 for a variety of purposes, including transaction tracking and zero-copy data transfer, which was described earlier. The service processor 275 also replaces the known default read, write and ioctl functions in the file_operations struct for the device file interface 510, which allows all messages to be intercepted and evaluated according to their header content. The service processor 275 uses a proc interface, which is provided by using the create_proc_entry function and replacing the file_operations and inode_operations structs of the proc file. The read and write functions are used to display and modify adjustable parameters within the service processor 275.

The running configuration 610 must be maintained in the service memory 270 in a searchable format, and the data structure of the running configuration 610 must allow a search function that is fast and scales less than linearly for a larger database. The data structure chosen has a tree format, which is a reduced version of a DOM tree. The configuration manager 265 maintains the stored configuration, and where the stored configuration is an XML document, the XML must be well formed. The configuration manager 265 supports a subset of XML comprising elements, attributes, and PCDATA and CDATA content.

The running configuration 610 is generated using the following data structures.

```
typedef struct leaf {
      void* data;
} leaf;

typedef struct element{
      void* children;
      struct tag* attributes;
      short c_num; /* number of children */
      short a_num; /* number of attributes */
} element;

typedef struct tag {
      char name[MAX_TAG_LENGTH + 1];
      element* content;
} tag;
<root>
    <child1>
          <grandchild>leaf text 1</grandchild>
    </child1>
    <child2 attr1='attr1 text' attr2='attr2 text'>leaf text
2</child2>
</root>
```

FIG. 8 shows the data structure 800 in the service memory 270. The root element 801 of the data structure 800 consists of a tag struct 802. The name of the root element 801 is stored in a name field 804, which has a predetermined maximum length. A content field 806 points to an element struct 808, which records the number of children

and attributes of the root element 801 in c_num 810 and a_num 812 fields. The element struct 808 also contains an attribute pointer 814, which points to an array of tag element attributes. This is a contiguous list of tag structs, one for each attribute of the root element 801. The children field 816 points to a similar list for the child tags of the root element 801.

Each child 821 and 822 and attribute 823 and 824 tag, has a pointer 826, 828, 830 and 832, respectively, to an element struct 834, 836, 838 and 840, respectively. These in turn, can point to further children or attributes. The children pointer 826 and 828 is a void* to allow for the case when the tag content is simply text, i.e. it is leaf tag 818 in the XML tree. The type of struct at the end of the children pointer can be guessed by the content of the c_num field. For example, if the c_num field is zero and the child pointer is not, then it is a pointer 852 to a leaf struct 854. The leaf struct 854 contains the actual data content of the tags, and the data is stored as a void* pointer.

The tag struct lists as pointed to by the children and attributes fields of the element struct are sorted by the service processor 275, and the sorting is based on an integer representation of the tag/attribute name, which is stored in the name field 804, as an example. The sorting algorithm used by the service processor 275 is Quick sort, as will be known by one skilled in the art

A comparison function is used by the service processor to determine the ordering of two tag structs, and the comparison function decides on the basis of the integer value of the tag name by taking sets of four consecutive bytes of the name string and returning the result of the integer comparison. If the integers comprising the four bytes being compared are equal, then the next four bytes are compared and so on. This results in a definite order for the task struct list, which makes it possible to search through the struct list in less than 0 (n) time using a searching algorithm such as binary

search. A binary search is used to achieve an O(log n) search time for a particular tag in a tag struct list. The comparison function used is the same as that used by Quick sort.

The running configuration 610, as stored in the mmaped service memory 270, is vulnerable to memory fragmentation and associated problems, such as a reduction in the speed of the search algorithm to handle non-contiguous storage of the data structures. Fragmentation occurs when there are additions or deletions to the running configuration 610. In the preferred embodiment of the present invention, a predetermined amount of fragmentation of the running configuration 610 is tolerated before any action is taken by the service processor 275, and an adjustable threshold tolerance level is provided via a proc interface.

The service processor 275 attempts to minimize the fragmentation of the running configuration 610. As the data structures that are more likely to cause fragmentation are the tag struct arrays, when the running configuration 610 is created, the tag struct arrays are packed in very closely in the service memory 270 to make the best use of the memory space of the service memory 270. In addition, the search algorithm that is used by the service processor 275 also requires that the tag structs remain in arrays. As a result, the insertion and removal of data from the running configuration 610 may result in expansion or contraction of the tag struct arrays. For example, an expansion will cause an entire array to be copied to another location to insert a new array element; and a contraction will reduce the size of the array. In both cases, a "hole" is created in the service memory 270. The service processor 275 tracks and reuses the "holes" created by such actions, and tries to reduce the occurrence of "holes" by allocating all the tag struct arrays together in contiguous memory space, away from the other structures in the running configuration 610.

The memory space required for the running configuration 610 may run out if the available free space and or the memory "holes" in the service memory 270 are not large enough to accommodate the running configuration 610. When this occurs, the service processor 275 signals the configuration manager 265 to attempt to defragment the service memory 270. When defragmentation does not result in sufficient free memory space to accommodate the running configuration 610, the service processor 275 allocates more memory space to store the running configuration 610.

The XPath implementation in the embedded system 200 of the present invention has to support a fast and scalable search algorithm. This is implemented using a string matching algorithm, which compares multiple characters at a time. A significant difference between the XPath implementation, in accordance with the present invention, and the original WWW XPath specification, is that the XPath implementation, in accordance with the present invention, does not support 'nodesets' as results. In other words, an XPath expression, in accordance with the present invention, that does not point to a single 'node' is taken to be invalid instead of returning a list of possible matches. This is to ensure that the application, such as the application 225, is aware of the exact parameter that is being changed, and receives a deterministic response.

The XPath implementation, in accordance with the present invention, interprets expressions of the following types:

1) Simple XPath expressions with just tag/attribute names e.g. interface/ip;
2) Positional expressions of the format tagname[position] e.g. interface[2]/ip;
3) Search expressions of the format tagname[@attribute='search string'] or tagname[.='search string']; and

4) XPath expressions with the following axis keywords, such as: parent or '..', sibling, child and node/self or '.'.

The XPath expressions are evaluated from left to right, and evaluation stops when an error occurs, or when there are multiple matches. Since the tags are ordered in the running configuration 610 as determined by the sorting algorithm, and not by the position of the tag in the file, an explicit ordering results. When there are multiple child tags with the same name, the order in which they are arranged in the tag struct list is determined as follows:

1) If both tags have no children (i.e. when c_num = 0), then the comparison is based on the content of the tag.

2) If both tags have children, then the comparison is based on the first child tag name. If that matches as well, then the comparison restarts with the first child of each tag, and continues deeper into the data structure until a decision can be made.

3) In either case, if a tag has no child/content, then that tag is considered 'smaller'.

The running configuration 610 is split into several tree data structures, which allows information in the running configuration to be grouped logically, as well as spread across different tree data structures, thus reducing the average depth of each tree data structure. As a result, searching the running configuration 610 is faster, since the number of comparisons performed is reduced.

With reference to FIG. 9, an ESI message 900 has a predetermined format, consisting of a header 905 and content or payload 910. The header 905 contains the following fields: message length 915, transaction ID 920, and message type 925. The corresponding C struct for the message header 905 is provided below.

```
typedef struct msg_hdr {
    char[4] length;
```

char[4] trans_id;
int msg_type;}

The transaction ID field 920 is left empty by applications, such as the application 225, when making a request. The service processor 275 fills the transaction ID field 920, which is returned piggy-backed on the response to the request. The message length consists of the most significant bits of the integer length value in length[0], and so on to length[3].

ESI messages can be of the following types: read, modify, insert, delete, poll, and event notification. The message content 910 consists of XPath expressions, and each message can contain multiple XPath expressions, which enable the modification or reading of multiple values in the running configuration 610. Further, the ESI messages can be of the following sub-types:

Read/Modify

a) Type 1 - The first field in the message is a context XPath expression, and the remaining fields are relative to that context. For a read message of type 1 the format will be as shown below.

context\0expression\0expression...

For a write message of Type 1 the format will be as shown below.

context\0expression\0value\0expression\0value\0

b) Type 2 - There are only expressions. The format of a read message will be as shown below,

expression\0expression\0...

and for a write expression will be as shown below.

expression\0value\0expression\0value

Insert

The only acceptable format is provided below.

expression\0

Only a single expression is allowed.

Delete

Type 1provide below.

context\0expression\0expression\0

Type 2 provided below.

context\0expression\0context\0expression\0

Event notification

a) Event register blocking - The format is provided below.

expression\0expression\0expression...

b) Event register signal - The format is provided below.

expression\0signalnumber\0expression\0signalnumber...

c) Event register passive - The format is provided below.

expression\0expression\0...

IPC register

Type 1 format is provided below.

context\0expression\0value\0expression\0value\0

Type 2 format is provided below.

expression\0value\0expression\0value\0

An IPC registry in the service memory 270 enforces a data structure with certain required fields in ESI messages. The XPaths expressions are checked against this prescribed data structure, after the general validity of the ESI message has been verified. The fields in the ESI messages are delimited using the NULL character i.e. '\0'. The last character in a message must be '\0'. The message type is a bitwise OR of constants, as provided in the following TABLE 1.

Message	Message type entry
Read	1
Write	2
Delete	4
Poll	8
Insert	16
Event notification blocking	32
Event notification signal	64
Event notification passive	128
Type 1 message	256
Type 2 message	512
Data in message	1024
Data through shared memory	2048

TABLE 1

A context XPath expression is normally used when there are multiple operations on a family of closely related nodes. For example, there may be the need for the following information in the running configuration 610 /interface/ip /interface/broadcast /interface/ netmask. Instead of sending the entire XPath expression each time in separate ESI messages, it is possible to send a 'context' expression of /interface and ordinary expressions of ip, netmask and broadcast. This reduces the amount of processing done within the service processor 275, and also the amount of data sent by an application, such as the application 225.

The service processor 275 caches the most recent contexts on a per application basis. If in place of an expected context string, the application sends an empty string, the cached context is used. In addition, whenever any simple expression does not begin with a '/' character, it is assumed to be relative to the cached context. An ESI message is considered to be complete in itself, and after processing a message, the service processor 275 performs any notification actions relating to any modified data. The application requesting a service must therefore ensure the completeness of the data, in order to not waste a service call.

In order to allow for multiple non-conflicting actions to occur within the embedded system 200 at the same time, every action

cannot be blocking. Consequently, once an application has made a service request, the service processor 275 must not stop responding to messages from other applications. A transaction based mechanism, which is provided by the service processor 275, is used to avoid an application having to wait for a change after issuing a service request before proceeding; especially if that change is known to take time. The transaction based mechanism tracks the state for every action that occurs within the embedded system 200.

An application that writes a message to the service memory 270 is allocated a transaction ID, which is implemented by a monotonically increasing counter. The application can obtain this ID by its next read from the device, and uses the transaction ID to refer to all further actions relating to this transaction, such as its status, its outcome, and data resulting from the initial message. The kernel maintains the status of each transaction, and can retrieve it using the transaction ID as the key. Every transaction in kernel space 110 results in the creation of a data structure to store the status of the transaction. A C programming language representation of this data structure is provided below.

```
typedef struct transact {
        int trans_id;
        leaf* data; /* leaf structure as above in the running configuration
*/
        char status;
} transact;
```

The status field is used to represent the following conditions:

- Success - the leaf struct will contain any data that was requested, or null.
- XPath error (the XPath expression was invalid) - the leaf struct will be null.

- Service error (the service implementation returned an error) - the leaf struct may contain an error message.
- Processing - the leaf struct will be null.
- No event.
- Activated event.

Every action taken on behalf of an application that needs a response from the service processor 275, is treated as a transaction. The transaction ID is made accessible to the application. The list of transact structs that are generated as transactions occur are stored in contiguous locations in a section of mmapable service memory 270. Completed transactions are deleted, and the transaction ID wraps when the maximum integer limit is reached.

As described previously, the event sub-system 285 provides different forms of notification mechanisms, which allow applications to chose the notification method that suits them most. The alternatives include a choice between explicit notification and application polling, as well as blocking and non-blocking versions.

Explicit notification is processed using kernel waitqueues or real-time signals, as will be known to one skilled in the art. Passive notification is processed by recording events in a transact struct. An application must request notification by sending a message consisting of the following:

Message type

- 'Event register blocking' for the waitqueue implementation.
- 'Event register signal' for the real time signal implementation.
- 'Event register passive' for the passive notification.

The message type must also indicate whether the application wants to receive the data through a message or read it itself using the shared memory space 505.

Message content

This is an XPath expression of the parameter that will generate an event when changed, followed by the Real Time signal number if requesting notification by that method. Real Time signals are a POSIX signal standard which allows for queuing of signals unlike normal UNIX signals. A Real Time signal is an asynchronous notification mechanism, which is used for inter-process communication (IPC). In addition to allowing signal queuing, Real Time signals also allow for piggy-backing of small amounts of data along with the signal notification.

Transaction Id

Empty

The kernel allocates a transaction ID for each event, and there are a range of transaction IDs reserved for particular events that are not used for normal transactions, which prevent ID clashes between event transactions and other non-event transactions.

In order to handle event transactions, the element struct is extended as shown below.

```
typedef struct element{
        void* children;
        struct tag* attributes;
        short c_num; /* number of children */
        short a_num; /* number of attributes */
        event* ev;
} element;

typedef struct event {
        void* notify;
        char not_type;
} event;
```

The new struct event is used to define the type of event that should be generated when a particular parameter in the running configuration 610 is modified. The field not_type can represent one of the following:

- Signal - notify is a pointer to task_struct of the application.

- Queue - notify is a pointer to a waitqueue, which is initialized with a call to init_waitqueue_head.
- Passive - notify is a pointer to the transact struct that is created for the newly allocated transaction-ID.

An event is said to have taken place when a value in the running configuration 610 has changed. When this occurs, the service processor 275 takes action on the basis of the not_type field of the appropriate event struct. The event struct is derived from the running configuration 610 as will be described below.

When the immediate parent element struct of the leaf data that is modified contains an event struct, that event struct is taken. Otherwise, the search algorithm traverses the data tree structure in reverse to find the closest ancestor with a registered event handler, and uses that one.

Based on the not_type field from the event struct, the following actions, as described below, are taken.

- Signal - a real-time signal is generated and sent using the send_sig_info kernel function to the process whose descriptor is in *notify.
- Queue - a wake_up_interruptible is called on the waitqueue head pointer in notify.
- Passive - the status field of the transact struct from the element struct is updated.

A more detailed operation of the embedded system 200 in accordance with the present invention, by way of example, will now be described. A component of the embedded system 200 must first categorize itself into one of a predetermined set of classes of ESI components. Based on this classification, the service provider 215 component makes use of the appropriate ESI facilities, in accordance with the ESI protocol to communicate with the other ESI components in the embedded system 200.

A service provider 215 component is typically implemented in a low level programming language such as C. The service provider 215 developer must first use predetermined ESI conventions to decide on the storage location of service provider parameters in the running configuration 610. Subsequently, the appropriate XML structure is designed and added to the stored conf XML files on disk.

When implemented, the first action taken by the service provider 215 on starting up, is to register for an event notification for a modification of its monitored parameters in the running configuration 610. The service provider 215 component typically requests an explicit notification in one of two ways.

- Signal notification – This is when a signal handler function uses sigaction and one of the Real Time signal numbers instead of the standard signal macros. Alternatively, a sigaction/sigprocmask system call could be used. Using signal notification, a message is written to the service processor 275, with type 'Event register signal', and with the body of the message containing the XPath expression of the service parameter in the running configuration 610.
- Queue notification – This is when a message with type 'Event register queue', and the appropriate XPath expression, is written to the device file in the service memory 270. The kernel then causes the application that sent the message to sleep on the associated waitqueue by using sleep_on_interruptible.

The choice between the two notification methods is dictated by the number of services offered by the service provider 215 component. If it offers just one service, it should use the waitqueue notification since it would be sleeping or inactive the rest of the time. For a service provider 215 component that provides multiple services, such a service provider 215 component can register for multiple signal notifications using a different Real Time signal number for each service.

When developing a service consumer 205, such as the application 225, a developer uses the ESI conventions to decide on how to request a service, and a first step is to devise the appropriate XPath expression to request the service. The application 225 is usually written in a high level language such as Java, and would not be able to use the mechanisms of the shared memory 505. When operating, the application 225 writes a message with type 'write', without specifying a transaction ID, to the service processor 275. The message payload should contain the appropriate XPath expression and new values for the parameters. The message type can also indicate whether the request should be blocking or non-blocking.

A non-blocking message returns immediately. The application 225 must then perform a read to the device file in the service memory 270, to obtain a message with a transaction-ID. The application then uses a 'poll' type message with this transaction ID to obtain the status of the transaction. The message payload returned by the service processor 275, contains the performance status and any data generated by the service request. A blocking call returns only when the service request is completed, either success or failure. The procedure to check the result is the same as that described earlier. The service processor 275 extracts the payload from the message written to it, assigns a transaction-ID to the newly started transaction, and creates a new transact struct for this transaction. The service processor 275 then notifies the configuration manager 265 through a signal mechanism, and the configuration manager 265 reads the XPath expression from the service processor 275 through a mechanism of the shared memory 505, and performs the search on the running-configuration 610. The configuration manager 275 then provides addressing information to the appropriate element struct or equivalent that provides return information, which is suitable for a wide variety of services.

The configuration manager 265 modifies the running configuration 610 on the assumption that the modifications will be accepted by the service provider 215 component, as is the case more often than not. The configuration manager 265 also keeps the old values of the data that is modified in order to be able to rollback the changes if necessary. Take for example, a section of the running configuration 610 as shown below.

```
<interface>
  <name>eth 0</name>
  <ip>10.0.1.10</ip>
  <netmask>255.255.255.0</netmask>
  <broadcast>10.0.1.255</broadcast>
</interface>
```

A message sent to the running configuration /interface/ip\010.0.1.11\0 causes the running configuration 610 to be modified to show as shown below.

```
<interface>
  <name>eth 0</name>
  <ip>10.0.1.11</ip>
  <netmask>255.255.255.0</netmask>
  <broadcast>10.0.1.255</broadcast>
</interface>
```

and the message to change to that shown below.

```
/interface/ip\010.0.1.10\0
```

Thus, the configuration manager 265 is able to rollback any changes if they do not take place as anticipated. The service processor 275 uses the information found in the event field of the appropriate element struct to notify the service provider 215 component. The notification can be in one of three ways as described earlier. On being

notified by the service processor 275, the service provider 215 component reads from the device 255, to obtain a message containing addressing information to the same element struct. The service provider 215 component then uses its knowledge of the running configuration 610 data structures to perform the required service action based on the data/parameters that have changed. Continuing with the same example, the service implementation will change the IP address of the interface with name "eth 0" to "10.0.1.10".

The configuration manager 265 as well as the service provider 215 component must update the transaction status. The configuration manager 265 updates the status from 'No event' to 'Event activated', assuming that there are no problems with the XPath expression. Otherwise, the configuration manager 265 sets the status to 'XPath error'. The service provider 215 component sets the status to either 'Service error' or 'Success'. Since the transact structs are available in the mmapable service memory 270, the configuration manager 265, and the service provider 215 component, can perform a linear search based on the transaction-ID and update the status field of the appropriate transact struct.

Earlier, a brief description of the service provider-consumer 217 component was provided, and now a more detailed description follows. As described the service provider-consumer 217 functions as both a service provider and a service consumer, and here a prior art application library 235 will be used as a part of an illustrative example for a service provider-consumer 217 component.

When the prior art application library 235 is to be used as a component in the embedded system 200, the art application library 235 is “wrapped” in an ESI shell 240. Services that were formerly provided by the application library 235 through an API in a prior art embedded system, are now be made available as ESI services in the embedded system 200. Hence, in this aspect the service provider-consumer 217 functions as a service provider to the applications,

such as the application 225. Although the ESI services that are made available by the service provider-consumer 217 to the applications, may be restricted relative to the services of the application library 235 in prior embedded systems, the ESI shell 240 advantageously shields the applications 225, from problems caused by the application library 235, as was described earlier in an example relating to memory allocation in the prior art.

As an ESI component, the service provider-consumer 217 is an active component, and can now also function as a service consumer. When functioning as a service consumer, the service provider-consumer 217 can register for event notification, and perform some activities, which in a prior system could only be performed in response to explicit function calls from an application. As an example, when an application, say application 225, requires that a file be streamed to a remote location, the application 225 sends a respective service request to the service provider-consumer 217. At this point the service provider-consumer is functioning as a service provider. The service provider-consumer 217 then functions as a service consumer and calls/uses the services of a network manager to effect streaming of the requested file.

More specifically, as a service provider, the service provider-consumer 217 would have registered to provide streaming services. When the application 225 requires streaming service, the application sends a request for streaming services, which provides the name of the file and the destination address for the streaming service. The service provider-consumer 217 then serves out the appropriate file to the destination. However, since the service provider-consumer 217 is dependent on the network manager for network configuration services, the service provider-consumer 217 also registers to receive notification when the network parameters change, as this would affect the streaming service.

Consider the example of a mobile communication device, which initially uses a General Packet Radio Service (GPRS) connection to a communication network. If the mobile communication device is also equipped with a wireless Ethernet (802.11b) interface, it would be advantageous for the mobile communication device to switch to a wireless Ethernet communication network when the mobile communication device comes within range of such a network, to take advantage of the higher bandwidth supported by the Ethernet connection. This network switching is performed automatically by the network manager component in accordance with the present invention, as it sets up the required IP interfaces and routes. The service provider-consumer 217 would then be notified via the ESI, at which point the service provider-consumer 217 switches to streaming over the newly configured Ethernet connection. Hence, the service provider-consumer 217 functions as a service provider to the application 225 and as a service consumer to a network manager.

ESI CONVENTIONS

Following on the above example, related ESI conventions are now described in relation to the streaming service. Here, the service provider-consumer 217 needs to know the best interface to use at a given time, when it receives a request for a streaming service. Conventions for ESI or ESI conventions that are adopted in the ESI XML tree structure of the running configuration 610 ensures that the service provider-consumer 217 knows the best interface. For example, the ESI conventions would require that all ESI compliant service providers provide services in the network section of the ESI XML tree structure of the running configuration 610, as shown below:

```
<esi>
    <network>
        <interfaces>
            <interface @name="eth0">
                <ip></ip>
                <netmask></netmask>
```

```
                    <broadcast></broadcast>
                </interface>
            </interfaces>
            <routes>
                <route>
                .....
                </route>
            </routes>
        </network>
</esi>
```

Such tree structures would are pre-defined and will be independent of the implementation of the actual service provider components for the respective parameters. Where a variety of service provider component developers develop a variety of service provider components, these service provider component developers are encouraged not to break the pre-defined hierarchy of the ESI XML tree structure of the running configuration 610.

Again, following the streaming example described earlier, the service provider-consumer 217 would expect to find the interface definitions at /esi/network/interfaces/interface in the ESI XML tree structure of the running configuration 610, and the service provider-consumer 217 then registers to receive notifications at that point in the ESI XML tree structure of the running configuration 610.

It is also possible to use search expressions such as /esi/network/interface/interface[@name="eth0"] to select the appropriate interface.

PASSIVE NOTIFICATION

Alternatively, the service provider-consumer 217 can register for a passive notification of changes in the interface setup of the ESI XML tree structure of the running configuration 610. Upon registration, the service provider-consumer 217 receives a transaction-ID on its next read from the ESI device. In order to check for an event at this location, the service provider-consumer 217 occasionally writes poll messages with the appropriate transaction-ID

to the ESI device, and on the next read from the ESI device, the event status (triggered/not-triggered) is sent back in the message, thereby providing notification.

The present invention as described provides a component-based embedded system that uses an ESI, which extends the concept of modularity to the object code level.

This is accomplished by a services model comprising service consumer components that use services; service provider components that provide the required services, and a service manager which performs the role of a central service authority that regulates the provision of the services between the service consumer components and service provider components. An extensible service interface (ESI) is defined, which enables communication between the service consumer components and the service manager component, and enables communication between the service provider component and the service manager component. The ESI provides an abstraction layer between the service consumer components and the service provider components, and the service manager component, where the service consumer components and service provider components do not need to communicate directly with each other. Instead, the service consumer components and service provider components need only communicate with the service manager component via the common ESI.

This advantageously allows the ESI components to be modified at run time, allowing a change in functionality of the components to be effected without the need for re-compilation or re-integration, thereby supporting a component-based system.

Thus, the present invention, as described provides a component-based embedded system and method therefor, which overcomes or at least reduces the abovementioned problems of the prior art.

It will be appreciated that although only particular embodiments of the invention have been described in detail, various modifications

and improvements can be made by a person skilled in the art without departing from the scope of the present invention.

Claims

1. An embedded system comprising:

at least one service provider for registering to provide at least one service, the at least one service provider for receiving a service request associated with the at least one service, for having the at least one service performed, and for providing a performance status indicating the performance of the at least one service;

at least one service consumer for communicating one of a plurality of service requests for the at least one service, and for receiving the performance status; and

at least one service manager being operably coupled to the at least one service provider and to the at least one service consumer, the at least one service manager for receiving the one of the plurality of service requests from the at least one service consumer and for providing the service request to the at least one service provider, and the service manager for receiving the performance status from the service provider, and the service manager for sending the performance status to the at least one service consumer.

2. An embedded system in accordance with claim 1 further comprising at least one service provider-consumer, the at least one service provider-consumer adapted to function substantially as the at least one service provider when the service request is received from the service manager, and the at least one service provider-consumer adapted to function substantially as the at least one service consumer when communicating the one of the plurality of service requests for the at least one service to the service manager.

3. An embedded system in accordance with claim 1 wherein the embedded system comprises at least one user space and at least one associated kernel space.

4. An embedded system in accordance with claim 3 wherein the at least one service provider comprises at least one service provider manager in the at least one user space, wherein the at least one service provider manager manages the at least one service.

5. An embedded system in accordance with claim 4 wherein the at least one service provider further comprises at least one service provider driver in the at least one kernel space, and wherein the at least one service provider driver is coupled to communicate with the at least one service provider manager in the at least one user space.

6. An embedded system in accordance with claim 5 further comprises at least one hardware space, wherein the at least one hardware space comprises at least one device for providing the at least one service, and wherein the at least one device is coupled to communicate with the at least one service provider driver in the at least one kernel space.

7. An embedded system in accordance with claim 3 wherein the at least one service consumer comprises at least a first service consumer component in the at least one user space, wherein the at least the first service consumer component is adapted to provide at least some of the plurality of service requests.

8. An embedded system in accordance with claim 7 wherein the at least one service consumer further comprises at least a second service consumer component in the at least one user space, wherein the at least the second service consumer component is adapted to communicate the one of the plurality of service requests for the at least one service, and is adapted to receive the transaction performance.

9. An embedded system in accordance with claim 3 wherein the at least one service manager comprises at least a first service manager component in the at least one kernel space and at least a second service manager component in the at least one user space.

10. An embedded system in accordance with claim 9 wherein the at least the first service manager component in the at least one kernel space comprises:

a service memory for storing at least one service configuration provided by the at least one service provider during registration, wherein the at least one service configuration comprises at least a notification specification and the performance status;

an event-subsystem for providing a service request notification in accordance with the notification specification to the at least one service provider;

a communication sub-system for receiving the one of the plurality of service requests from the at least one service consumer, and for receiving a request for the performance status from the at least one service consumer and sending an updated performance status to the at least one service consumer; and

a service processor coupled to the service memory, event-sub-system, and the communication sub-system, the service processor for facilitating storage of the at least one service configuration from the at least one service provider in the service memory, for monitoring the at least one service configuration in the service memory for updates to the performance status, and in response to the communication sub-system indicating that the request for the updated performance status from the at least one service consumer has been received, the service processor providing the updated performance status to the at least one service consumer through the communication or event sub-systems.

11. An embedded system in accordance with claim 10 wherein the at least the first service manager component in the at least one user space comprises a configuration manager coupled to the service memory for managing a plurality of service configurations, wherein the at least one service configuration comprises one of the plurality of service configurations.

12. A method for an embedded system comprising the steps of:

a) providing at least one service provider, at least one service consumer, and at least one service manager being operably coupled to the at least one service provider and to the at least one service consumer;

b) the at least one service consumer communicating a service request for a particular service to the at least one service manager;

c) the at least one service manager communicating the service request for the particular service to the at least one service provider;

d) the at least one service provider communicating a notification to the at least one service manager; and

e) the at least one service manager communicating the notification to the at least one service consumer.

13. A method in accordance with claim 12 further comprising the step of the at least one service provider registering to provide the particular service, prior to step (c).

14. A method in accordance with claim 13 wherein the step of registering comprises the step of the at least one service provider communicating a registration message to the at least one service manager.

15. A method in accordance with claim 14 wherein the step of communicating the registration message comprises the step of specifying at least one of a plurality of types of notification.

16. A method in accordance with claim 13 further comprising, after the step of registering, the step of the at least one service manager determining the particular service is provided by the at least one service provider.

17. A method in accordance with claim 12, prior to step (d), comprises the step of the at least one service provider providing the particular service.

18. A method in accordance with claim 12, comprises the step of the at least one service provider providing the particular service during step (d).

19. A method in accordance with claim 12, prior to step (d), comprises the step of the at least one service provider determining the particular service cannot be provided.

20. A method in accordance with claim 12, wherein step (b) comprises the step of the at least one service consumer amending data of the at least one service manager.

21. A method in accordance with claim 20, wherein step of amending comprises the step of addressing the data.

22. A method in accordance with claim 21, wherein step of addressing comprises the step of providing a predefined XPath expression.

23. A method in accordance with claim 12, wherein step (c) comprises the step of the at least one service manager determining that data associated with the at least one service provider has been amended.

24. A method in accordance with claim 23 further comprising the step of the at least one service manager communicating to the at least one service provider that data associated with the at least one service provider has been amended.

25. A method in accordance with claim 24 wherein the step of communicating to the at least one service provider that data associated with the at least one service provider has been amended comprises the step of notifying the at least one service provider using a pre-selected one of a plurality of types of notification.

26. A method in accordance with claim 12, wherein step (b) comprises the step of the at least one service consumer communicating a notification request to the at least one service manager.

27. A method in accordance with claim 26, wherein the step of communicating a notification request comprises the step of specifying that a notification be communicated when change occurs to data of the at least one service manager.

28. A method in accordance with claim 27, wherein determining when change occurs to data of the at least one service manager comprises the step of the at least one service manager determining that data of the at least one service manager has been amended.