1 Introduction

Inter process communication (IPC) is a set of methods for exchanging information among multiple processes. It can be provided by the operating system kernel, such as pipe, message queue on Unix / Linux, or it can be implemented by third party library using infrastructures of certain operating system, such as MPI. The communication parties can reside on one physical machine, such as those using Unix domain socket, or they can span over multiple machines, such as those using TCP/IP socket. IPC provides the foundation of segmenting large program into pieces of loosely coupled modules, which improves programmer productivity and quality of software in several facets, such as isolation for security and parallelism for speed.

However, common low level IPC mechanism treat the information being conveyed as a simple byte stream, or simple datagrams of bytes, which is cumbersome to use in complex programming problems. Object oriented IPC (OO-IPC), such as D-bus[1] and CORBA[2], serves as a modern middleware for communication between different software components, especially in event-driven model application. With OO-IPC programmers can focus on the core logic of their application and do not need to deal with low level signal handling and message parsing. As a result, desktop applications with GUI and server application running in distributed environment utilize OO-IPC heavily.

The basic idea of OO-IPC is encapsulating every software components into objects with well defined interface, and abstracting every communication among them as remote procedure call (RPC) and event delivering (which is just a special RPC that does not need return value). In this RPC model, callee (also named “server”) is the program that implements the actual function of a method, and caller (also named “client”) is the program that invoke the method. Usually methods is defined inside interface, and interface is implemented by objects. Objects have predefined path so as to be located easily. To make a remote procedure call of a certain interface, caller must invoke this call through a proxy or broker that represents this interface (sometimes called client side proxy). This proxy or broker translates the called methods to a message and send it to the remote side through traditional IPC mechanism or network. The translation usually needs to introspect the remote object model and marshal all the argument to a byte stream. On the remote side, when such message is received, it is recognized by the message protocol handler and all the arguments are de-marshaled back to objects. Then usually a dispatcher is used to deliver this remote procedure call to the corresponding adapter or broker of this interface (sometimes called server side proxy), and finally the adapter or broker call the correct method of callee's objects. After the call, return value is returned in the same
way to the caller. Sometimes one just need to tell others that something happened. Such event is usually represented by a special case of method call, whose return value is not needed. The basic scheme of OO-IPC is shown in the figure below.

![Figure 1: OO-IPC basic scheme](image)

Because OO-IPC usually defines platform independant abstract interface and wire message protocol, the caller and the callee can be implemented by different object model, in different language, on different system and processor. The proxy / adapter or object broker is usually implemented in libraries, and programmer only needs to define their own object and interface, and implements these objects and interfaces. Usually there is a interface definition language (IDL) used in a certain type of OO-IPC, such as XML. The topology of communication connection can be a software bus which support multicast, or a point to point network. Such underlying topology is usually transparent to the programmer.

## 2 D-Bus

D-bus is the most widely used OO-IPC mechanism on desktop linux. It is designed to replace DCOP[3] in KDE and Bonobo[4] in GNOME. It defines a standard protocol that facilitates the communication among desktop applications, GUI widgets, and deliver kernel hardware event (by udev[5], formerly HAL[6]). D-bus supports both remote procedure call (RPC) and signal-like event, with enforced authentication and security. D-Bus also defines its own interface definition language, which is a special XML restricted by D-Bus document type definition (D-Bus interface DTD). Since D-Bus is designed to support different programming language, it defines a platform independant type system, which can be easily mapped to native type system of every object oriented programming language.

### 2.1 D-Bus architecture

The basic architecture of D-Bus is shown in the figure below. D-bus has three basic parts: the library (libdbus), the bindings[7] (libdbus-glib, libdbus-qt, Java binding, Python binding, etc.), and the daemon (dbus-daemon[8]). The library provides the basic data structure and low layer operation of bus management, session and protocol management, authentication, and message transportation. It is implemented in C and it does not deal with object model and event-driven model. Usually user application use higher level bindings instead of using the library directly. Such binding layer handles object mapping, object marshaling / de-marshaling, and event loop. The daemon is used to manage the bus and route message. There are two kind of daemons, system wide daemon (per machine), and session daemon (per login session). The former is usually used for deliver event generated by kernel (hardware status event), or sent to kernel
The topology of D-Bus looks like a software bus shared by every process connected to it, so that any process can communicate with another process, or multicast / broadcast an event. This topology is implemented in a hub-spoke way, i.e. all processes connect to the daemon. Any process running on D-Bus has a connection to the bus with a unique bus address. Some applications acquire human readable bus names, which is called well-known bus names, so that other applications can use their services (i.e. call their methods), deliver events to them (i.e. send a signal), and listen events from them (i.e. add a match rule to listen to a signal). Well-known bus names and unique bus address work like domain name and IP address, and the resolution is handled by the daemon. For example, the message bus itself has a bus name of “org.freedesktop.DBus”, which can be regarded as the daemon’s bus name. Usually bus name has the form of reversed domain name, for the sake of eliminating the possibility of name collision.

Under each connection, a process can define multiple objects, and each object has a unique object path starting from the root path (i.e. “/”). Object path in one bus name is independent from that in another bus name. Usually in each well-known bus name there is an object with the path that is just the same as bus name, which serves as the main object of this application. For example Linux power controller has a well-known bus name of “org.freedesktop.UPower”, and an object with the path “/org/freedesktop/UPower” under this bus name. Object can implement several interfaces, which is a collection of methods, signals, and properties. Interfaces in D-Bus can be easily mapped to abstract class in C++ or interface in Java. The method is mapped to the method of object, the property is mapped to the field of object, and signal is mapped to special methods used as sending signal helper and signal handler. For example, the main
object of power controller implements an interface named “org.freedesktop.UPower”, and there are several methods in it such as “Hibernate()”, “Suspend()”, etc. As the method name suggests, if you call “Suspend()”, the computer will be suspended. There are several standard interfaces in D-Bus. One is “org.freedesktop.DBus.Introspectable”, which has only one method named “Introspect()”. This method returns a string contains the object’s interface definition in XML. Another is “org.freedesktop.DBus.Properties”, which has three methods named “Get()”, “GetAll()”, and “Set()”. These methods are used to get and set the properties of an object. The bus names, objects and interfaces can be observed by a GUI tool named D-Feet[9]. Here is an example of how the objects and interfaces are organized for the power controller.

D-Bus support two basic IPC operations - signal and method. Signal implements a publisher / subscriber model and it is usually used to deliver certain event to other application. Method implements an one-to-one caller / callee model, which is the same as RPC. All operations will eventually generates bus messages and be transported through Unix domain socket. Message is routed by the daemon. For method call, routing is based on source and destination. For signal, routing is based on various match rules on the attributes of message, such as message type, bus name, object path, interface name, etc.

2.2 D-Bus performance
However, there are some problems that have not been addressed. First, many event-driven system and application need certain real-timeness feature, which Linux and D-Bus does not provide. Second, in many cases D-Bus binding infrastructure such as GKT[10], QT[11] and Java are not easily and usually employed in embedded systems. So new libraries, lightweight language bindings, event driven middleware and application framework are more desirable. Third, and the most important, current D-Bus implementations has significant overhead in embedded environment with restricted CPU speed and memory space. The overhead is fourfold. One is abstraction overhead caused by multiple layers of encapsulation in the library (four layers, including transport layer, message protocol layer, object layer, and event layer). Another is security overhead caused by argument assertion and validation, safe string operation, safe memory allocation and freeing. Then another is object model overhead caused by serializing and deserializing (memory copy), introspecting (XML parsing), and method name binding (D-Bus interface v-table access and indirect call). The last one is routing overhead in the daemon caused by matching, filtering, policing, and forwarding. One RPC may cause four context switches because the calling message first wakes up the daemon for routing and then the callee, and the return message dose two wakeups again. A performance data of the echo client server is listed below, obtained by Linux perf tools. This test is made using just libdbus (without C++ object layer overhead) with 1 million method calls. The calling message argument is a 5 byte character string and the return value is a 4 byte status and a 4 byte integer, which is typical for D-Bus message call. In our test, the average delay of one RPC is about 1.22 ms (on a Core 2 Duo E8400 with 4GB memory), which is considerably long.

<table>
<thead>
<tr>
<th>process</th>
<th>average cpu cycles for one RPC</th>
<th>total CPU usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>client (user)</td>
<td>58,148</td>
<td>10.9%</td>
</tr>
<tr>
<td>client (kernel)</td>
<td>19,118</td>
<td>3.5%</td>
</tr>
<tr>
<td>server (user)</td>
<td>53,955</td>
<td>10.2%</td>
</tr>
<tr>
<td>server (kernel)</td>
<td>17,475</td>
<td>3.3%</td>
</tr>
<tr>
<td>daemon (user)</td>
<td>344,265</td>
<td>64.8%</td>
</tr>
<tr>
<td>daemon (kernel)</td>
<td>38,229</td>
<td>7.2%</td>
</tr>
<tr>
<td>total</td>
<td>531,190</td>
<td>100%</td>
</tr>
</tbody>
</table>

### 2.3 D-Bus usage

Despite those problems, nearly all desktop GUI applications on Linux use D-Bus, simply because those are not problems for desktop machines and traffic over D-Bus is really light. These applications include basic desktop environment components such as GNOME panel and
session manager, most desktop environment management gadgets such as network manager and display manager, crucial system utilities such as Nautilus[12] file system manager and screen saver, device applets such as bluetooth applet, system controller such as UPower[13] and UDisk[14], kernel component configuration interface such as PolicyKit[15], hardware event monitor such as udev, and most GUI user applications such as Audacious music player and Brasero disk burner[16].

Another important usage of D-Bus is delivering hardware state information to user space applications. Hardware event is usually generated by device drivers inside the kernel, which uses the uevent[17] interface to send these events to udev[5]. The systemd[18] daemon (formally udevd) interprets these information and translates them to predefined D-Bus signals, and broadcast it on D-Bus. Any processor that is interested in hardware event can add a match rule so that such signals will be listened. Using this publisher / subscriber way, the hardware event interface become more flexible.

Although D-Bus is mainly used as OO-IPC mechanism in desktop environment, one can try to port or implement it in embedded environment which is not suffered from those problems mentioned before. With further middleware, one can build event-driven application framework for certain embedded systems, especially those systems which needs interactive communication or using sensor / actuator model, such as robotics. Magnenat and Mondada proposed such a scheme on robots called ASEBA[19]. Sarker and Dahl also designed a communication system based on D-Bus named HEAD[20]. Such system are implemented on embedded linux and peripheral MCUs.

3 A simple example of echo client and server using D-Bus

Here we use a simple example of echo client and echo server to demonstrate the usage of D-Bus in C++. Also we dig a little deeper and briefly introduces the internal implementation of D-Bus. Codes are listed in appendix A.

```xml
<node name="/org/freedesktop/DBus/Examples/Echo">
  <interface name="org.freedesktop.DBus.EchoDemo">
    <method name="Echo">
      <arg type="s" name="input" direction="in"/>
      <arg type="s" name="output" direction="out"/>
    </method>
    <signal name="Echoed">
      <arg type="v" name="value"/>
    </signal>
  </interface>
</node>
```
This the interface definition written in XML. It has a three layer hierarchy of node, which is the D-Bus object, interface, and method or signal. Argument is declared inside method, with specified type, name, and direction. The direction can be in or out, which means input argument or return value. To support various programming language, D-Bus uses an independent type system. Types include static primitive types such as integer, floating point number, character and string, static compound types such as structure and array, and a special dynamic type variant. A variant can be regarded as a union of all the primitive types, and it is determined at run time which primitive type is used. These types can be mapped into C++ easily, and even some structure types can be mapped to STL container. To represent the independent type system, every type in D-Bus has a unique signature of one ASCII character. For example in the interface definition above, method Echo() accepts a string whose signature is ‘s’, and the Echoed() signal has an argument of type variant whose signature is ‘v’.

In this interface, there is one method named “Echo()” and one signal named “Echoed”. The echo server uses an adapter which implements this interface and implements the actual Echo() function. Adapter makes the server object accessible from other process. The echo client uses a proxy which also implements this interface and implements the actual signal handler Echoed(). Proxy delegates the RPC and message handling and makes RPC like just method call on local objects.

Using the following shell script one can generate the adapter class and the proxy class. Here dbusxx-xml2cpp is a tool that translate the interface definition in XML to C++ code. If “--proxy” is used, the proxy class is generated in the specified header file. The proxy class serves as a template for the client class. It defines all the methods in the interface and implements them as RPC. Also if “--adaptor” is used, the adapter class is generated in the specified header file. Like proxy class, adapter class serves as a template for the server class. It defines all the methods in the interface, but leaves them unimplemented as abstract methods. So the server class extends the adapter class and actually implements them.

```
#dbusxx-xml2cpp echo-introspect.xml \  
  --proxy=EchoClient.h \  
  --adaptor=EchoServer.h
```

The client class extends the EchoDemo_proxy clas. All the programmer need to do in the client class is implementing the signal handler and finish the main function. Signal is always emitted by the adapter and received by the proxy, just the same as method is always called by the proxy and returned by the adapter. Note that in the main() function, a connection to the session bus is obtained at the beginning and the client object is initialized using server’s bus name and object path. The client object is a proxy of remote server object. This proxy object has the exact same interface of the remote server object, so as that in the interface definition. So in the greeter_thread(), the Echo() method of the proxy is called, and the proxy translates it to a RPC message on D-Bus and send it to the remote adapter. Then the proxy get the returned message and translates it to a local string, and return it as the Echo() method of the proxy returns. This Echo() method is implemented in the proxy class.
The server extends the adapter. The main() function works similar as that in the client. The only difference is that server can only responds to method calls. The server class must implement the abstract methods declared in the adapter, which are those methods defined in the interface. Here the Echo() function is implemented. Sometimes server may emit signals to the client, which can be achieved by calling the Echoed() function of the adapter. For example here in the server “this->Echoed()” is called inside the Echo() function of the server class. The Echoed() function is already implemented in the adapter class, so does Echo() in the proxy class.

After the initialization, the main() function of client spawn another thread, and called the enter() method of dispatcher, which enters the main loop inside the library. Similar thing happened in the main function of server. The only difference is that server just respond passively, so it does not need the thread for making call or sending signal. Inside the D-Bus library the main loop does three things: write buffered message to the bus, read message from the bus and put it to the local message buffer, and dispatch the read message to the right handler. The dispatching uses a hash table to bind method and signal names to the actual method and signal handler function. These names are given by the interface definition written in XML, and the methods and handler functions are predefined in the proxy or adapter class generated by the tools shown above. Such functions in the proxy and adapter are abstract methods that must be implemented by the client or the server class. Once implemented, they are bound by the v-table the object, so that the name of methods can be correctly resolved for the dispatcher. In this way, it is actually the dispatcher in the main loop that calls the corresponding method. If the method blocks, it will block the whole dispatcher and no further message buffered will be processed. To prevent such case happening, D-Bus C++ binding can utilize multiple dispatchers or dispatcher with multiple threads.

4 CORBA

The Common Object Broker Architecture (CORBA)[2] is another object-oriented IPC. It is first released in 1991 by Object Management Group (OMG)[21] in the hope of helping the seamless communication among processes that are written in different languages and run on different machines. It utilizes an Interface definition language (IDL) to define the interfaces that the so called “object request brokers” (ORBs) uses to present to the other processes that written in other programming languages. In this architecture, a machine can either be a server who provides the ORBs for other to call, or a client who uses the methods of ORBs provides by the server or both. Currently, most of the commonly used programming languages have standard mapping to IDL. These languages include Java, Python, C, C++, C++11, Lisp, Ruby, Smalltalk, Ada, COBOL, PL/I. there are also languages that support non-standard mapping such as Perl, Visual Basic, Erlang and Tcl. One thing worth to mention is that the easiness of mapping from the original programming to IDL are not at the same level. For object-oriented languages such as Java and Python, because of the closeness of the language nature, the mapping is quite natural and easy. But for language such as C, which doesn’t implement the object-oriented, the mapping is quite complicated.
4.1 CORBA architecture

Next, let’s take a close look at the architecture of CORBA to explore its principal mechanism. Here is a diagram below shows the overview of the CORBA architecture in a very concise way.

![CORBA architecture diagram](image)

**Figure 4: CORBA architecture**

On the client side, a client is the process that request to use the operations (methods) of an ORB object through its reference, the IDL stub works as a proxy that converts the method calls of the client into messages.

On the server side, the IDL skeleton works as an adapter that converts messages generated by IDL stubs into method calls. The servant on the server is the entity that actually implements the request on the object. Then the return value / error would be transformed by the IDL skeleton and send back to the client via the ORBs.

The messages formed through IDL stubs and received by IDL skeleton are transmitted through a layer called “ORB Core” and this layer actually provide the message passing through the standard communication protocol such as General Inter-ORB Protocol (GIOP) and Internet Inter-ORB Protocol (IIOP). These protocols defines how the ORBs should pass information back and forth and with this message passing layer, distributed programs that works on different computers can communicate with each other over the internet.

5 A simple example of CORBA usage
The following is a simple example of two Java programs utilizing CORBA to do the inter-process communication on one computer[22]. In this example, the server side implements two simple methods, `sayHello()` and `shutdown()`. As their names infer, the former return a string “Hello world!” and the latter close the server program. The client program will call these two methods and get the “Hello world” string and then shut the server down. Implementing this example requires three main steps: defining and compile the remote interface, implementing the server, and implementing the client.

5.1 Define and compile the remote interface

No matter what languages the clients and servers are using, the remote interfaces are the same and therefore have to be written in IDL. For this example, the interface is for two method and it is defined in the following table. Note that the exact implementations of these methods are not here.

<table>
<thead>
<tr>
<th>Hello.idl</th>
</tr>
</thead>
<tbody>
<tr>
<td>module HelloApp</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>interface Hello</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>string sayHello();</td>
</tr>
<tr>
<td>oneway void shutdown();</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

With this IDL file, different languages have different compiler to translate this interface to the corresponding programming language. In Java, the compiler that JDK 7 provides is called idlj and the translated result is shown in the following table. Along with the package and interface translation, other files such as `HelloPOA.java`, `_HelloStub.java`, `HelloHelper.java`, `HelloHolder.java` are also generated by the compiler idlj.

<table>
<thead>
<tr>
<th>Hello.java</th>
</tr>
</thead>
<tbody>
<tr>
<td>package HelloApp;</td>
</tr>
<tr>
<td>public interface Hello extends</td>
</tr>
<tr>
<td>HelloOperations,</td>
</tr>
<tr>
<td>org.omg.CORBA.Object,</td>
</tr>
<tr>
<td>org.omg.CORBA.portable.IDLEntity</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>
5.2 Implementing server

At the server side, the major implementations include

- Creates and initializes an ORB instance;
- Creates a servant instance (the implementation of one CORBA Hello object) and tells
  the ORB about it;
- Get object reference of ORB from the servant and the root naming context;
- Registers the new ORB object under the name "Hello";
- Waits for invocations from the client;

5.3 Implementing client

At the client side, the major implementations include

- Creates and initializes an ORB object;
- Obtains a reference to the ORB object with the name "Hello";
- Invokes the object's sayHello() and shutdown() operations and prints the result;

For detailed java code of 2 and 3, please refer to appendix B.

6 Implement Kademlia with OO-IPC

In this part a more complex example is introduced, a variant of Kademlia[23] distributed hash
table (DHT)[24] implemented with OO-IPC. DHT is a hash table which is partitioned and
separated among a lot of nodes, with a key of fixed length binary number and a value of any
serializable object. Every node has a unique id, and it holds a portion of information of the
whole hash table, whose key has the same bitwise length of id. In our variant, id is just an 32
bit unsigned int (standard Kademlia use 160 bit SHA hash), so as the key, and value is any
string. The information on a node is usually has the key “near” its id. To define how near it is
(i.e. the distance), a metric method is used, which is exclusive or (XOR) in Kademlia. Besides
the partial hash table, there is a special structure called k-buckets on each node, which is stores
the information of other nodes known by this node. Each k-bucket stores at most k nodes, and
there is at most n buckets at all, where n is the bit length of node id (or the length of key, which
is the same). Here k is a global parameter, which is set to 4 (standard Kademlia use 20 for k).
The ith k-bucket stores the nodes information whose distance is between $2^i$ and $2^{i+1}$. The nodes
information is a doublet of <node address, node id>. The address is bus address for D-Bus and IP address and port number for CORBA. For example, the 6th bucket stores at most 4 nodes with distance smaller than 128 and bigger than 64, which means any node’s id is different from this node’s id in the 6 least significant bits. Nodes in one bucket is sorted in a LRU list. In such an organization, a node always has more knowledge on the nearer and newer nodes. Here is diagram of k-buckets structure of a node in standard Kademlia.

![Diagram of k-buckets structure of a node in standard Kademlia](image)

Any Kademlia node is an object implements KadNode interface, which has four methods. The first is Ping(), which is used to check liveness. The second is Store(), which instructs the node to store a <key, value> pair in its local portion of hash table. The third is Delete(), which instructs the node to delete a <key, value> pair. Standard Kademlia does not include this method. The fourth is FindNode(), which takes a key as an argument and returns the information of at most k nodes whose id is closest to the key. The last is FindValue(), which behaves like FindNode() but it returns the value if the requested node has the value corresponding to the key in local portion of hash table. Here is the definition of the interface.

```xml
<?xml version="1.0" ?>
<node name="/edu/asu/cse438/Kad">
  <interface name="edu.asu.cse438.Kad">
    <method name="Ping">
      <arg type="i" name="ping" direction="in"/>
      <arg type="i" name="pong" direction="out"/>
    </method>
    <method name="Store">
      <arg type="i" name="key" direction="in"/>
      <arg type="s" name="value" direction="in"/>
      <arg type="i" name="return" direction="out"/>
    </method>
    <method name="Delete">
      <arg type="i" name="return" direction="out"/>
    </method>
  </interface>
</node>
```
All the operations such as search, insert, delete, must be conducted at a point of view of one node, because all the information is distributed on all nodes.

To search a key value pair starting from one node (local node), a program works as below is needed.

1. Get alpha nodes whose id are closest to the key from the k-buckets of the local node;
2. Initiate alpha parallel FindNode() RPC on these alpha nodes. These nodes will return alpha times k more nodes, which many of them might be duplicated. The returned nodes composed a set R;
3. Pick alpha nodes whose id is closest to the searching key from R, and repeat 2, until no closer node is found. Because the k-buckets of node always have more nodes that is close to itself, these steps will always converge.
4. now R contains the nodes whose id is closest to the searching key globally. Initiate alpha FindValue() RPC on them. If any of them return the value, the search succeeds. If none of them return the value, the search fails and there is no such key value pair in the DHT.

This searching algorithm often discards half the number of nodes on the searching path, which makes it efficient. In this algorithm, alpha is a global parameter like k. If one node can issue multiple RPC, i.e. if it can do RPC in multiple thread (because RPC will always block the caller), alpha can be greater than 1 and RPC is paralleled. Otherwise these RPC is serialized. In this example we just use 1 (standard Kademlia use 3). Also some bookkeeping is needed in the searching. When nodes are returned as set R, these nodes are also added to the k-buckets. If a k-buckets is full, nodes are replaced according to LRU. Standard Kademlia will keep a first seen time for each known node, and call Ping() on the oldest node in LRU list. If it is alive, it is moved to the head of the LRU list. If all the nodes are alive, the node with the lastest seen time is kicked out of the k-bucket. This is based on the empirical result on P2P network that if one node is online for a longer time than another node in the past, it will keep online for a longer time in the future. Here we do not use such complex methods in this example. We just replace nodes with LRU. If one RPC fails because the lost of remote connection, the target node is
simply kicked out of k-buckets or the set R. Here is a diagram illustrating searching in Kademlia.

![Diagram illustrating searching in Kademlia](image)

On calling method of a remote object, one must first create a corresponding proxy object. This proxy object is created dynamically (proxy object is statically created in the example of echo client server), and this proxy object is keep as long as the node is still in the k-buckets of the local node. When it is kicked out, its proxy object is released.

The insert operation and delete operation behave similarly. First we locate alpha nodes globally closest to the key, and use Store() to insert a <key, value> pair, or use Delete() to delete a <key, value> pair. If alpha is larger than 1, one <key, value> pair is stored alpha times redundantly. If a previous entry with the same key exists, it is replaced. Due to the churn of nodes' liveness in the network, old entries might remain in some condition. So in this example each entry has a timestamp. If inconsistent entries is found in the process of search, a Delete() is issued to the node that has the old entry.

When a node joins the DHT, it must obtain some information of nodes by other ways other than the DHT itself. So we have some permanent “well-known” nodes who have “well-known” addresses and never go offline. The new node will first fill these well-known nodes in its k-buckets (i.e. “get to know” these nodes), then its k-buckets are updated on every search. For the sake of efficiency and fairness for all the nodes, the addresses of these well-known nodes must be “well” distributed. In our example, there are 32 nodes with ids that are evenly distributed from 0 to $2^{31} - 1$. 

When a node exits the DHT, it does not need to “tell any other node”, i.e. make any RPC on any node. It is designed in Kademlia for the sake of robustness and fault tolerance, so as the alpha times redundancy of the entries in different nodes. An “elegant” exit can be implemented by telling all the nodes in the local k-buckets to flush the information of the local node. But this increases traffic with nearly no gains, because the offline node will always be flushed when the first time a RPC is called on it. And the k-buckets will not be polluted by the offline nodes because at every replacement, a Ping() RPC is used to check liveness. However, in our example such pollution does exist. But since we do not consider failure of the nodes in our example, the pollution is also ignored.

Our Kademlia implementation even has a value when all the nodes are located in the same machine connected through D-Bus. Consider all the entries are distributed among different process. One can use a broadcast search on the shared bus by broadcast a signal, and the node that has the entry will reply in unicast by a RPC, like ARP over Ethernet. If all the nodes remains silent, after a timeout, the search is regarded as failure. The complexity of this broadcast search is O(1), but it wakes up all the nodes in the bus. If there are a lot of nodes on the bus, since all nodes need to grab a piece of CPU time, context switches and checking the signal on every node in such a flood search will kill the performance. On the other hand, Kademlia nearly always wakes up log(n) nodes for a search, which is very efficient.
References

[6]. “http://www.freedesktop.org/wiki/Software/hal”
[7]. “http://www.freedesktop.org/wiki/Software/DBusBindings”
[9]. “https://live.gnome.org/DFeet/”
[10]. “http://www.gtk.org/”
[12]. “https://live.gnome.org/Nautilus”
[16]. “http://projects.gnome.org/brasero/”
[19]. Stephane Magnenat, Francesco Mondada, “ASEBA Meets D-Bus: From the Depths of a Low-Level Event-Based Architecture into the Middleware Realm”
[22]. “http://docs.oracle.com/javase/1.4.2/docs/guide/idl/jidlExample.html”
[23]. Petar Maymounkov and David Mazieres. Kademlia: A Peer-to-peer Information System Based on the XOR Metric
// echo-client.cpp

class EchoClient :
    public org::freedesktop::DBus::EchoDemo_proxy,
    public DBus::IntrospectableProxy,
    public DBus::ObjectProxy
{
public:
    EchoClient(DBus::Connection &connection, const char *path, const char *name)
        : DBus::ObjectProxy(connection, path, name) {} 
    void Echoed(const DBus::Variant &value) { cout << "!" }
};

void *greeter_thread(void *arg) {
    cout << g_client->Echo("Hello~") << endl;
    return NULL;
}

int main() {
    DBus::Connection conn = DBus::Connection::SessionBus();
    EchoClient client(conn,
        "/org/freedesktop/DBus/Examples/Echo",
        "org.freedesktop.DBus.Examples.Echo");
    pthread_create(&thread, NULL, greeter_thread, (void *) 0);
    dispatcher.enter();
    pthread_join(&thread, NULL);
    return 0;
}

// echo-server.cpp

class EchoServer :
    public org::freedesktop::DBus::EchoDemo_adaptor,
    public DBus::IntrospectableAdaptor,
    public DBus::ObjectAdaptor
{
public:
    EchoServer(DBus::Connection &conn)
        : DBus::ObjectAdaptor(conn, "/org/freedesktop/DBus/Examples/Echo") {} 
    std::string Echo(const std::string &str) { this->Echoed(str); return str; }
};
Appendix B: Hello client and server on CORBA in Java

// HelloServer.java

import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;
import java.util.Properties;

class HelloImpl extends HelloPOA {
    private ORB orb;

    public void setORB(ORB orb_val) {
        orb = orb_val;
    }

    // implement sayHello() method
    public String sayHello() {
        return "\nHello world !!\n";
    }

    // implement shutdown() method
    public void shutdown() {
        orb.shutdown(false);
    }
}

public class HelloServer {

    public static void main(String args[]) {
        try{
            // create and initialize the ORB
ORB orb = ORB.init(args, null);

// get reference to rootpoa& activate the POAManager
POA rootpoa = POAHelper.narrow(
    orb.resolve_initial_references("RootPOA"));
rootpoa.the_POAManager().activate();

// create servant and register it with the ORB
HelloImplhelloImpl = new HelloImpl();
helloImpl.setORB(orb);

// get object reference from the servant
org.omg.CORBA.Object ref = rootpoa.servant_to_reference(helloImpl);
Hello href = HelloHelper.narrow(ref);

// get the root naming context
org.omg.CORBA.Object objRef =
    orb.resolve_initial_references("NameService");
// Use NamingContextExt which is part of the Interoperable
// Naming Service (INS) specification.
NamingContextExtncRef = NamingContextExtHelper.narrow(objRef);

// bind the Object Reference in Naming
String name = "Hello";
NameComponentpath[] = ncRef.to_name( name );
ncRef.rebind(path, href);

System.out.println("HelloServer ready and waiting ...");  
// wait for invocations from clients
orb.run();
} catch (Exception e) {
    System.err.println("ERROR: "+ e);
e.printStackTrace(System.out);
}

System.out.println("HelloServer Exiting ...");

// HelloClient.java

import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
public class HelloClient
{
    static Hello helloImpl;

    public static void main(String args[]) {
        try{
            // create and initialize the ORB
            ORB orb = ORB.init(args, null);

            // get the root naming context
            org.omg.CORBA.Object objRef =
                    orb.resolve_initial_references("NameService");
            // Use NamingContextExt instead of NamingContext. This is
            // part of the Interoperable naming Service.
            NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

            // resolve the Object Reference in Naming
            String name = "Hello";
            helloImpl = HelloHelper.narrow(ncRef.resolve_str(name));

            System.out.println("Obtained a handle on server object: "+ helloImpl);
            System.out.println(helloImpl.sayHello());
            helloImpl.shutdown();
        } catch (Exception e) {
            System.out.println("ERROR : "+ e);
            e.printStackTrace(System.out);
        }
    }
}