Abstract:

Different Medium Access Control (MAC) protocols for Wireless Sensor Networks (WSN) use different strategies for handling collisions, idle listening, overhearing and control packet overhead to tradeoff performance (delay, bandwidth utilization, throughput) for energy cost efficiency. As there is no single best protocol, different network topology and application assertions may be used to guide the MAC design.

In this work, we introduce a configurable architecture for Wireless Sensor Networks Medium Access Control. We allow applications to build a custom-made MAC protocol by selecting communication type, listening mechanism, duty-cycle and collision detection and handling mechanisms. We present our MAC configuration mechanism, and case studies presenting sample configurations featuring commonly used MAC protocols.

To demonstrate the framework's aptitude of the configuration architecture, the POP-C++ runtime libraries, developed in an internship in the School of Applied Sciences of Fribourg, are used as an application.
Discussion:

Wireless Sensor Networks have, by definition, very little processing, memory and energy resources to rely upon. This has created a very active field of research and a wide range of low power media access control protocols, each with their own strategies to maximize performance (delay, bandwidth utilization, throughput) while minimizing energy consumption.

These protocols' media access policies can generally be classified in 3 groups: CSMA-based, Slotted and TDMA-based.

CSMA based protocols use carrier sensing to evaluate if any other nodes in range are transmitting, then transmit if the channel is free. An example of CSMA based protocols is B-MAC, a protocol developed at Berkeley that uses a periodic clear channel assessment algorithm to evaluate when the channel is busy not only for transmission, but also for reception purposes.

Slotted protocols work similarly, except the carrier-sensing and posterior transmission only occur in the beginning of a fixed window of time. This allows for smaller duty cycles, since a node has the guarantee that no one is transmitting outside that time window. An example of a slotted protocol for WSNs is S-MAC, a protocol that uses small SYNC packets to broadcast the sleep/wake schedule of the nodes, in a way that the nodes around it can imitate and follow the same schedule. To account for clock skew, these SYNC packets are sent not only in the beginning of the network's use, but in its whole lifetime.

Finally, TDMA-based protocols use network-wide synchronized time frames which are divided into slots. These slots are then scheduled by an access point to be used by certain nodes for transmission and reception. An example of TDMA protocol for WSNs is L-MAC, a protocol which eliminates the need for an access point through the use of a bitmap that keeps track of which time slots are busy and which ones are free. Every node, upon reception of such bitmap, makes a logical OR operation with its own bitmap, to maintain consistency of the slot schedule across the network.

Due to the very heterogeneous nature of WSN applications, each of these protocols may excel in some scenarios and be completely unusable on others. It's up to the protocol designer to choose what end of the bargain is more appropriate for the application. For example, Low Power Listening, a CSMA-based protocol, performs very well if very low latencies are required (since the packets are sent as soon as they're available), but collapses under high loads since a very high number of collisions is certain to occur in such conditions.

Therefore, to allow for greater configurability and code reuse in WSN MAC protocols, we propose an infrastructure that, at compile time, lets the protocol designer make choices about how the protocol will behave at various points,
from the way each node decides if it can send data to the way it should handle collisions. In choosing which points of our MAC architecture would be configurable by the designer, we kept the following goals in mind: making sure that the widest possible range of MAC protocols could be "assembled" using our system and defining configurable points that, when combined, compose a complete MAC protocol, meaning that no parts of the protocol should be left out of the configuration, except for those common to all of them. With that in mind, we identified the following configurable points in WSN MAC protocols:

**Duty Cycle:** The duty cycle configuration dictates when the radio is allowed to operate. In simple CSMA-based protocols, the radio is always able to start transmitting or sampling the channel for a transmission. In slotted protocols, though, this period is limited to the active part of the defined time slot. In TDMA-based protocols, a node's duty cycle encompasses the Traffic Control slot, its uplink and downlink slots, and the contention period.

**Collision Avoidance:** The Collision avoidance measures in WSN MAC protocols may be comprised of Request to Send (RTS) and Clear to Send (CTS) packets, or a more complex Clear Channel Assessment algorithm. The RTS/CTS exchange serves as a way to avoid collisions because any nodes that can hear the CTS (and therefore are in range of the receiver) will refrain from transmitting and may even turn off their radios during the transmission of the packet. CCA, on the other hand, does away with the packet exchange overhead by having the transmitting node sense for any current transmissions before seizing the medium. There must also be the option for no Collision Avoidance, because the overhead from RTS/CTS packet exchange or CCA algorithms may be unacceptable in certain conditions, especially when the data payloads are very small.

**Collision Detection:** WSN MAC protocols use Acknowledgement packets to signal the sender that the packet was received. In high load situations where packet loss isn't an issue (dense networks where sensor information is bound to be redundant over at least two nodes, for example), it may be a smart choice to turn off acknowledgments and save the energy that is spent in their transmission.

**Collision Handling:** When a collision is detected, a protocol may choose to retransmit or simply increment a collision counter to let the application know of the amount of packets that have been lost. In the case of retransmission, backoff algorithms are used to minimize the chance of yet another collision occurring. These algorithms may range in complexity from random time before retransmission to the binary exponential backoff procedure that is found in 802.11: when the sender fails to receive its ACK, it chooses a random time within a Collision Window to retransmit. If the packet is lost again, the length of the Collision Window is doubled to reduce the chance of yet another collision. It's important to note that this configurable point is very dependent
on the enabling of Collision Detection.

These are the parts of the MAC protocol we found that could be independently implemented and assembled together to form a complete protocol. When transmitting, a node should observe the Duty Cycle, Permission to Send, Collision Avoidance, Collision Detection and Collision Handling configurations, in that order. When receiving, the node should observe the Duty Cycle and, after receiving a packet, the Collision Detection configurations.

The interface to the configurable features is through a traits system, where each configurable feature is represented by a variable. The user may then attribute a value to each of the variables to select the actual implementation to be used. One feature that could be implemented in the future is consistency checking, since, for example, using collision handling when collision detection is disabled not only makes the final code size unnecessarily bigger, but it might cause the protocol to malfunction.
Methodology:

The project will be realized in iterative steps, with the possibility of a new step cycle being executed:

1. Bibliographic Revision - In this step the mostly used MAC protocols for Wireless Sensor Networks are reviewed
2. Domain Engineering – Taking the reviewed MACs into account, find the configurable points
3. Reverse Engineering of the POP-C++ system – understand and document the POP-C++ runtime system for future use
4. Porting of POP-C++ - Reimplementation of the POP-C++ runtime system on EPOS running on WSN, for use as an exemplar WSN application
5. Modelling of the System – Modelling of the configuration architecture and definition of how will the protocols be configured
6. Load Tests – Realization of tests of POP-C++ running on WSNs with various workloads and communication patterns, to determine the correctness of the architecture
7. Report – Writing of the final report, describing the problem, the system and related works in detail
Conclusion:

This work has the goal of creating the necessary infrastructure for the easy configuration of Medium Access Control protocols for Wireless Sensor Networks. According to the proposed chronogram, all the software support for a complete protocol to be configured, generated and put to work quickly and with ease will be implemented.
Bibliografía:


Philip Levis, Sam Madden, David Gay, Joseph Polastre, Robert Szewczyk, Alec Woo, Eric Brewer, and David Culler. The emergence of networking abstractions and techniques in tinyOS. In First Symposium on networked system design and implementation (NSDI04), pages 1-14, San Francisco, California, USA, 2004.


Annex A:
Documentation generated from reverse engineering the POP-C++ runtime system
Section 1: OBJECT CREATION:

1. Upon a 'new()' call on the Integer object (the interface side, created by the parser), the 'interface::Allocate()' method is called.

2. The 'interface::Allocate()' method then uses 'interface::LocateResource()' to find a proper machine to run the job. If a target machine is configured in the OD, LocateResource() only checks for the availability of the binary code for that platform, using the CodeMgr object. Otherwise, it's going to use the JobMgr interface to find a proper machine.

There are three alternatives in finding a JobMgr:
   ● configuration through the od object (od.joburl field)
   ● configuration through the environment variables (reached through popcsystem::jobservice)
   ● localhost (popcsystem::GetHost())

Once the target JobMgr is found, the JobMgr::AllocResource() method runs the resource discovery function to find a node that satisfies the object's processor/memory requirements.

3. Depending on the output of LocateResource(), one of two functions is called:
   ● interface::LocalExec: when the object is to run on a determined node (not only localhost, this also includes the cases where od.URL is specified)
   ● JobMgr::ExecObj when the object is to run in a resource found by the JobMgr

4. After the allocation of the the object, the interface::Bind() method is called, creating the combox necessary for the interface to communicate with the broker.

5. Code flow returns to the Integer constructor, and the _Construct() method is called. This sends a message to the broker side, using the combox/buffer created in bind, telling it to create the actual object (invoke the real C++ constructor).

To illustrate the possible cases, observe the following 3 simplified sequence diagrams:
1) In this case, the constructor included a specific hostname on the od. Because of this, no JobMgr is involved in the instantiation of the object, and the method used to instantiate the new process is interface::LocalExec().
2) In this case, the constructor specified minimum processing power and memory requirements, so the JobMgr was used. MatchAndReserve() is the method responsible for comparing the requested resources with those that are available in this machine. In this diagram, the local machine was capable of running the instantiated object, so AllocResource() returned right away. To perform the actual instantiation of the new process, the JobMgr::ExecObj() is called. JobMgr::Exec() is semantically equivalent to interface::LocalExec(), in that it instantiates the new process.
3) This is the case where the first queried JobMgr's local machine doesn't have enough resources for the requested object; the first call to MatchAndReserve() (no. 3 in the sequence) returns false, causing a call to JobMgr::Forward() (not pictured for legibility). What this does is forward the AllocResource() call to a list of registered JobMgs. The second call to MatchAndReserve() then returns true and the accesspoint for the second JobMgr is returned all the way to the Integer interface. The following calls to ExecObj(), Bind() and _Construct() serve the same purpose as in the previous example, the only difference being that they are all directed to the second JobMgr, where the object will be run.
Section 2: METHOD CALLS:

All method calls in the interface side generate the creation of a message, using the combox/buffer functionality to account for encoding. The broker object (which is generated for each parclass during the parsing stage of the compilation) runs a thread, whose behaviour is in broker::ReceiveThread. It is basically a receive loop that looks for incoming packets, then uses a simple table lookup to execute them. The semantics of the POP-C++ parclass methods (sync, async, seq, mutex, conc) is also taken care of by the broker mechanism, before the actual function call occurs.

The sequence, then, is as follows:

1. The broker object for the parclass (in this case called Integer__parocobjBroker, a name given by the POP-C++ parser) enters the loop looking for messages from the interface

2. The function call in the interface side generates a message, packed in a buffer according to the correct format

3. The message leaves the combox in the interface side and reaches the broker, its header containing the name of the class, name of the method, and semantic value.

4. Upon receiving the message, the broker invokes the broker::RegisterRequest method, which accounts for the semantics of remote function calls

5. The Integer__parocobjBroker::Invoke() method, created by the parser, contains a table indexing all the methods of the Integer__parocobj class, and with a switch decides what method to call. It's important to note that the parser has renamed all Integer's methods to names indexed by declaration order, i.e.: Invoke_Integer_10, Invoke_Integer_11 and so on.

6. After calling the Invoke_Integer_[number], the actual method in the object (broker::_obj, a pointer to the actual object) is executed

7. If the method has a return value, a message containing it goes back to the interface side, and is unpacked and returned on the same function call that started the whole process, closing the cycle

To illustrate this, observe the following simplified sequence diagram:
The first event in this diagram is a call to Recv() on the buffer of the broker side. This call is generated by the loop in broker::Run(). Eventually a call to the Get() method being generated in the Application, then gets transformed into a message in the buffer (see the explanation of the implementation of Get() in the 'interface' class documentation). Immediately after this, the interface calls Recv() on the buffer, waiting for the return of the method.

As soon as the message crosses the network and reaches the broker, Recv() (no. 1 in the sequence) returns. The broker then unpacks the information in the message (method index, invocation semantics) and uses Invoke() to call the 'wrapper' method (Invoke_Integer_14(), in this case). The 'real' call to the actual object occurs, and the return value is sent back to the interface side using the same buffer. When the packet arrives in the interface, it unpacks the return value and the interface call to Get() returns.
Section 3: CLASS BY CLASS DOCUMENTATION

Class: interface

General description: The POP-C++ parser generates an interface class (that retains the original name) for each parclass it processes. For example, for a parclass A, its interface class will also be called A. The job of the interface class is twofold:

1. Finding a node in which the object can execute and creating a connection between itself and that object's broker
2. Packing method parameters/return values and transmitting them to the broker of the object where the methods will be executed

When processing a parclass, the parser makes some modifications to the class declarations. For example, the Integer test (included with POP-C++’s distribution package) goes from:

```cpp
parclass Integer
{
    public:
        Integer(int wanted, int minp) @{ power= wanted ?: minp;};
        Integer(string machine) @{ od.url(machine);};
        ~Integer();

        seq async void Set(int val);
        conc int Get();
        mutex void Add(Integer &other);
        mutex void Add([in] Integer &o1, [in] Integer &o2);

        async conc void Wait(int t);

        conc int Sum([in] int x[5000]);

    private:
        int data;
};
```

To:

```cpp
class Integer : virtual public interface
{
    public:
        Integer(int wanted, int minp);
        void _Construct(int wanted, int minp);

        Integer(string machine);
        void _Construct(string machine);

        void Set(int val);
```

int Get();
void Add(Integer &other);
void Add(Integer &o1, Integer &o2);
void Wait(int t);
int Sum(int x[5000]);

public:
  virtual const char *ClassName() { return "Integer"; };
Integer(const accesspoint &p) { Bind(p); };
Integer(const interface &inf) { SetOD(inf.GetOD());
  Bind(inf.GetAccessPoint()); };
Integer(const object *obj) { Bind(obj->GetAccessPoint()); };

As can be seen, the interface class has the same signature as the parclass, only without the
POP-C++ extensions to the language. This way, a regular C++ compiler such as GNU's G++
can compile it.

There are 3 major differences between POP-C++ code and the C++ code the parser
generates for the interface:

1. Embedding of computing requirements inside the constructor body
2. Embedding of the method invocation semantics inside the message header
3. Inclusion of inheritance to 'interface' and a few 'helper' methods.

All POP-C++ markup (method call semantics, node processing capabilities requirements) is
then transferred to the body of the method calls, and gets executed in runtime. This
document will now describe these 3 differences in more depth.

**Difference 1)**
The node processing requirements are represented in the final code as values in the object
descriptor, which is an object that every interface has (inherited from class 'interface'). The
first thing that the constructor of every interface object does is set the requirements in that
structure. This can be seen here, in the implementation of the first constructor of class
Integer (the interface of the original parclass, also called Integer):

```
Integer::Integer(int wanted, int minp)
{
  od.power(wanted, minp);
  Allocate();
  _Construct(wanted, minp);
}
```

**Difference 2)**
The method invocation semantics are embedded in the header of the message that will be
sent to the broker, to cause the actual execution of the method. Each of the flags that can
be attributed to the methods (sync, async, seq, mutex, conc) has a value that is logical
'OR'ed in the 3rd field of the message header. The possible values are these (found in broker_popc.h):

```
#define INVOKE_ASYNC 0
#define INVOKE_SYNC 1
#define INVOKE_ASYNC 2
#define INVOKE_CONSTRUCTOR 4
#define INVOKE_CONC 8
#define INVOKE_MUTEX 16
```

For example, in the following method, declared in the parclass:

```
conc int Get();
```

This method has both the 'conc' and the (implied) 'sync' flags, representing the method invocation semantics. Both of them will become part of the header of the message that is to be sent to the broker. This becomes clear by looking at the following snippet of code, extracted from the implementation of the Get() method in the interface side:

```
int Integer::Get()
{
    /* ... */
    message_header __buf_header(CLASSUID_Integer,14,9, "Get");
    /* ... */
}
```

The first parameter is the classid (CLASSUID_Integer), the second is the methodid (14) and the last is the method name ("Get"). The third, the semantics value, must be a sum of the possible values. Therefore, this method is 'conc', value of 8, and 'sync', value of 1, 1 'OR' 8 is 9, the value in the message header.

**Difference 3)**

The final difference is the inheritance of the interface class, and the additional constructors. The inheritance adds to the 'clean' parclass all the other fields and methods necessary for its integration with the rest of the POP-C++ runtime support. The additional constructors are nothing but ways to instantiate an interface to an already existing object; if for example node A created an Integer object, other interfaces could be instantiated for it, passing as a parameter to the constructor either an accesspoint (string representation of an network address), a pointer to an interface to be copied or an actual pointer to said object.

Now, for the actual class called 'interface', which every generated class will extend: these are the most important methods.
Main methods:

- **LocateResource()** - this method selects the target host of the actual object and then makes a reservation, through a series of decisions: first, if upon the object construction the user specified a hostname, then LocateResource() will just check for the existence of the object code for the specified platform and return. If not, it will use the JobMgr ( AllocResource() method ) to find a node in which to execute the object.

- **Allocate()** - After using LocateResource() to determine 'where' the object will be executed, this method uses either interface::LocalExec() or JobMgr::ExecObj() to instantiate the process with the object. If LocalExec() is used, the object will be run using SSH, and if ExecObj() is used, then the JobMgr will take care of instantiating the process. In either case, after this process the Bind() method will be called, to create the connections/comboxes necessary for method calling.

- **LocalExec()** - Similarly to JobMgr::ExecObj(), this method checks to see if it is to run in localhost or another host (through SSH), checks if it has to download the object file (using the script provided with the POP-C++ distribution, /services/webrun). After this it then builds the parameters of the new process and forks/execvp the object image. Once the process is run, LocalExec() waits for a message containing the accesspoint of the created objects.

- **Bind(accesspoint)** - cycles through the possible protocols, calling the other signature of Bind() to do the actual communication. If the user specified a certain protocol, it starts with that. Wraps the real Bind().

- **Bind(char *)** - creates a combox and a buffer ('__combox' and '__buffer', both attributes of class 'interface') and then uses combox::Connect to create the connection between the interface and the broker.

- **AddRef()/DecRef()** - add and remove a reference to the object, for garbage collection purposes. These methods construct a message, and send it through the combox created by bind. This will create a message, which will be treated by the broker::ParocCall() method.

**Note to Developers!!**
The following 2 classes are probably the parts of POP-C++ that need the most work. To send a simple packet you need a Buffer, a Connection AND a Combox, working together in a very unclear way. Wouldn't it be better if the Combox had a buffer or buffer factory of its own, and kept track of who it's connected to?

A proper analysis of the cardinality relations between these 3 classes must be carried out, but from what I've seen this is bad modelling that is simple to fix.

**Class: buffer**

**General description:** A buffer is the way POP-C++ enc/decodes information to ensure cross-platform consistency. It is meant to be extended and enable several ways of packing data, from raw send-as-it-is (buffer-raw) all the way to XML structuring (buffer-xml, found in /modules/xml/).

**Main methods:**

- **Pack()** - insert data into the buffer in a platform-independent way
- **UnPack()** - remove data from the buffer
- **Send()** - Puts a header on whatever data is packed in itself and uses the combox/connection it receives as parameter to actually send it

And the following are currently only used in the case of XML buffers, otherwise are just empty methods with no effect on the buffer:
Push() - create a new section in the complex structure
Pop() - remove a section from the complex structure

Class: combox (and connection)

General description: The combox is the workhorse behind all runtime communication in POP-C++. All method calls and return values go through two comboxes, one on the interface side and one in the broker side. The connection class keeps track of who the combox is connected to.

Main methods:

- Wait() - begin waiting on a new connection
- Connect() - connect to a certain url (ip address/hostname of the combox that is waiting for a connection)
- Send() - receives a pointer to a data section and its length, then iterates through it sending it to the connection's target.
- Recv() - reads len bytes from the socket to the provided pointer

Class: broker

General description: The broker is the interface to the actual object, and it is the broker who unpacks the messages that come from the network and carries out the actual function calls on the target object. In the same fashion as with the interface objects, there is one broker object created for each parclass that the POP-C++ parser processes, using the following name scheme: [Original class name]__parocobjBroker. The difference is that the broker class definition and implementation is generated almost from scratch by the parser, utilizing only the method names and other information extracted from the original parclass.

To make the actual call to the method code, each broker has a pointer to its object, called _obj. To maintain polymorphism, the type of this pointer is 'object *', and for that reason a dynamic cast must be performed each time the function call is carried through.

Using the earlier example, for the same Integer parclass as before, this is the broker class generated. To facilitate comprehension, a comment to the right side of the method declarations has been added, to associate these methods with the originals in the parclass:

class Integer__parocobjBroker: virtual public broker
{
    public:
        virtual bool Invoke(unsigned method[3], buffer &__brokerbuf,
                            connection *peer);

    protected:
        // constructor that takes computing power as param.
        void Invoke_Integer_10(buffer &_buf,
                                connection *__interface_output);
        // constructor that takes hostname as param.
        void Invoke_Integer_11(buffer &_buf,
                                connection *__interface_output);
For each of the methods that the parclass originally had (2 constructors and 6 regular methods), the parser created a 'stub', indexed by the order of declaration. These methods will be called from the newly created method, called Invoke(). This method uses a simple switch structure to 'redirect' function calls from the ReceiveThread to its other methods (Invoke_Internal_{index}). The implementation of Invoke() is the following, quite simple to understand:

```cpp
bool Integer_parocobjBroker::Invoke(unsigned method[3],
                                      buffer &__brokerbuf, connection *peer)
{
    if (*method==CLASSUID_Integer) switch(method[1])
    {
        case 10: Invoke_Integer_10(__brokerbuf, peer); return true;
        case 11: Invoke_Integer_11(__brokerbuf, peer); return true;
        case 13: Invoke_Integer_13(__brokerbuf, peer); return true;
        case 14: Invoke_Integer_14(__brokerbuf, peer); return true;
        case 15: Invoke_Integer_15(__brokerbuf, peer); return true;
        case 16: Invoke_Integer_16(__brokerbuf, peer); return true;
        case 17: Invoke_Integer_17(__brokerbuf, peer); return true;
        case 18: Invoke_Integer_18(__brokerbuf, peer); return true;
        default: return false;
    }
    return false;
}
```

Each of these methods, Invoke_Integer_[index], is responsible for calling the actual methods in the "real" objects, and to do so, it:
1. unpacks the parameters  
2. calls the method in _obj using the parameters  
3. saves the return value of the method (if available)  
4. packs the return value  
5. sends it back to the caller, using the same buffer it received in (buffer has a combox, which has a connection, so the return value always goes to the correct node)

As illustration, here is the implementation of 'Invoke_Integer_14()', which corresponds to 'Get()' on the original parclass:

```cpp
void Integer__parocobjBroker::Invoke_Integer_14(buffer &__buf,  
connection __interface_output)
{
    int _RemoteRet;
    Integer__parocobj * _obj=dynamic_cast<Integer__parocobj *>(obj);
    _RemoteRet=_obj->Get();
    if (__interface_output!=0)
    {
        __buf.Reset();
        message_header __buf_header("Get");
        __buf.SetHeader(__buf_header);
        __buf.Push("_RemoteRet","int", 1);
        __buf.Pack(&_RemoteRet, 1);
        __buf.Pop();

        if (!__buf.Send(__interface_output))
            exception::throw_errno();
    }
}
```

Since 'Get()' has no parameters, the method doesn't need to unpack anything before calling the method in the '_obj' reference. After saving the return in _RemoteRet, it packs it in the buffer that was passed as a parameter, and then sends it back to the interface side. All the other methods are similarly implemented.

**Main methods:**

Note to Developers!!  
The 'garbage collector logic', meaning the point in execution where the reference number is checked, is inside the GetRequest() method. Doesn't it make more sense to have it explicitly in Run()?

- Run() - This method initializes the structures in the broker, then enters a while loop calling GetRequest() and ServeRequest(). When GetRequest() returns false, Run() exits the loop and destructs the object and itself.  
- GetRequest() - used to get the next request in the request fifo. Blocks waiting for a new one if the fifo is empty. Returns false if the reference count reaches zero or isRunning turns false.  
- ServeRequest() - Calls DoInvoke() and takes care of the method call semantics in the
case the method is a 'mutex' method.

- **DoInvoke()** - Finally, this method makes the call to the Invoke() method, which has the switch mechanism explained earlier.

- **ReceiveThread()** - This thread is a loop that receives requests (using ReceiveRequest()) and calls ParocCall(). In the case ParocCall() returns false, it means that the requested method is a normal class method, and should be inserted in the 'request_fifo'. For that, it calls RegisterRequest(). There is one of these created for each combox.

- **ReceiveRequest()** - This method is called by the ReceiveThread(), and is used to 'translate' network messages received from the combox into 'request' structures for insertion in the fifo. If the method is 'async', this method responds with an 'ack' right away.

- **ParocCall()** - Method that handles internal POP-C++ methods (Add/DecRef, GetEncoding, BindStatus, Kill, ObjectAlive). If the method ID is higher than 10, this method assumes it's a parclass method and returns false.

- **RegisterRequest()** - This method is called by the ReceiveThread and either runs a method (in case it's a 'conc' method. The method is run using the 'invokethread' class, that creates a thread, calls DoInvoke() and then destructs the thread) or adds it to the request_fifo, an attribute of the broker. The requests in the 'request_fifo' will be subsequently handled by the loop in broker::Run().

### Class: object

**General description:** So far, both sides of the connection, the interface (caller) and the broker (callee), have been described. The only vital part of a POP-C++ application that hasn't been described yet is the actual code, contained in the 'object'. All the actual code in every parclass gets inserted, without a single modification, in a class named according to the following scheme: [Original class name]__parocobj. This class is simply a copy of the original class, without the POP-C++ markup (so that a regular C++ compiler can understand it), and an inheritance to object (so that the broker class can be generic, keeping a pointer to 'object' instead of 'Integer', for example).

To illustrate, our Integer parclass' parocobj is:

```cpp
class Integer__parocobj  : virtual public object
{
  public:
    Integer__parocobj(int wanted, int minp);
    Integer__parocobj(string machine);
    ~Integer__parocobj();
    void Set(int val);
    int Get();
    void Add(Integer &other);
    void Add(Integer &o1, Integer &o2);
    void Wait(int t);
    int Sum(int x[5000]);
  private:
    int data;
};
```
And the implementation of 'Get()' is:

```c
int Integer__parocobj::Get()
{
    return data;
}
```

**CLASS: JobMgr**

**General description:** The JobMgr is where all the code related to resource discovery, resource reservation, node tree structure and actual process instantiation is located. It is a parclass itself, so some of the methods are called remotely (the most important ones being the AllocResource() and ExecObj() sequence).

**Main methods:**

*Note to Developers!!*

The following overloaded declarations of AllocResource and MatchAndReserve should be renamed to different methods.

- AllocResource(signature w/o ‘fitness’) - this method makes a data structure, called requestInfo, to call the method with the same name but other signature. Wraps the real AllocResource.

- AllocResource(signature w/ ‘fitness’) - this method performs a few checks: first, if the code is available for the local platform (CodeMgr::QueryCode()). then it matches the required processing power and memory (MatchAndReserve()). In the case of a failed local allocation, it calls the Forward() method to find another JobMgr that can host the object.

- MatchAndReserve(signature w/o ‘inoutfitness’) - this method performs the breaking down of a multiple object AllocResource call (howmany > 1), and calls the other MatchAndReserve() method to do the actual processor/memory requirement checking. Wraps the real MatchAndReserve.

- MatchAndReserve(signature w/ ‘inoutfitness’) - this method checks for each of the requirements of the object, and calculates a 'fitness' value based on the available resources. In case of a successful reservation of resources, it returns the ID of the reservation that was made. It's worth noting that the object isn't constructed on the broker side until the _Construct() method is called on the interface.

- Forward() - this method traverses the list of known nodes (a NodelInfoMap called ‘neighbors’, an attribute of JobMgr). It takes the requested number of objects that have a fitness smaller than 1 (unfit to run in this machine) and calls the AllocResource() method on the child node, sending only the object count it couldn't run locally as a parameter. These call forwardings also have a 'iptrace' mechanism, that takes a note of all the nodes this resource request has reached. This is used to avoid an infinite loop in resource allocation, and time wasted re-checking a node for a resource we already know it doesn't have.

- Exec() - calls fork() and execvp() to create the process for the new job.

- ExecObj() - builds the parameter string for the Exec() call, then returns the accesspoint for each of the created objects so that the interface knows where to bind and finally send the method calls.
**Class: CodeMgr**

**General description:** The CodeMgr keeps a database of the compiled objects, which platforms they are compiled for and where the actual code file is located. Through QueryCode(), it does a simple lookup procedure to verify availability of a certain object on a given platform.

**Main methods:**

- **RegisterCode()** - Adds an entry to the database, receiving object/platform/codefile values.
- **QueryCode()** - Receives an object name, a platform, and puts into the string 'codefile' the location of the code file. This may be a remote address (ftp:// or http://) or an absolute path in the local machine.
- **GetPlatform()** - Receives the name of an object as parameter, and puts in the string 'platform' the platforms for which that object is available.

**File: infmain_popc.std.cc**

**General description:** This file is where the main() function of all POP-C++ interface code comes from; The main() method that the user defines in his POP-C++ application is renamed to 'parocmain()', and gets called by the actual main() defined here. This main() has the purpose of processing command line parameters, and then calling the actual main() of the application.

**File: objmain_popc.std.cc**

**General description:** This file is where the main() function of all POP-C++ broker/object code comes from; its purpose is to process command line parameters, initialize the broker for the object, and call broker::Run().