Everything About Ether

Ether is a smart, high speed and lightweight messaging system. Not until the later stages of Galore design that it was realized to have a module like Ether is a necessity. Flexibility is the core aspect of the Galore architecture, there by making all the components act like services (having well defined interfaces). But, the question remains, as how these components interact with each other; The conventional methods of SOAP/RMI are either very slow or heavy weight. Ether module(earlier called as streams) is proposed to address these issues. Ether is completely developed in-house and will be implemented with an idea of making all the galore components communicate faster.

Requirements(functional)

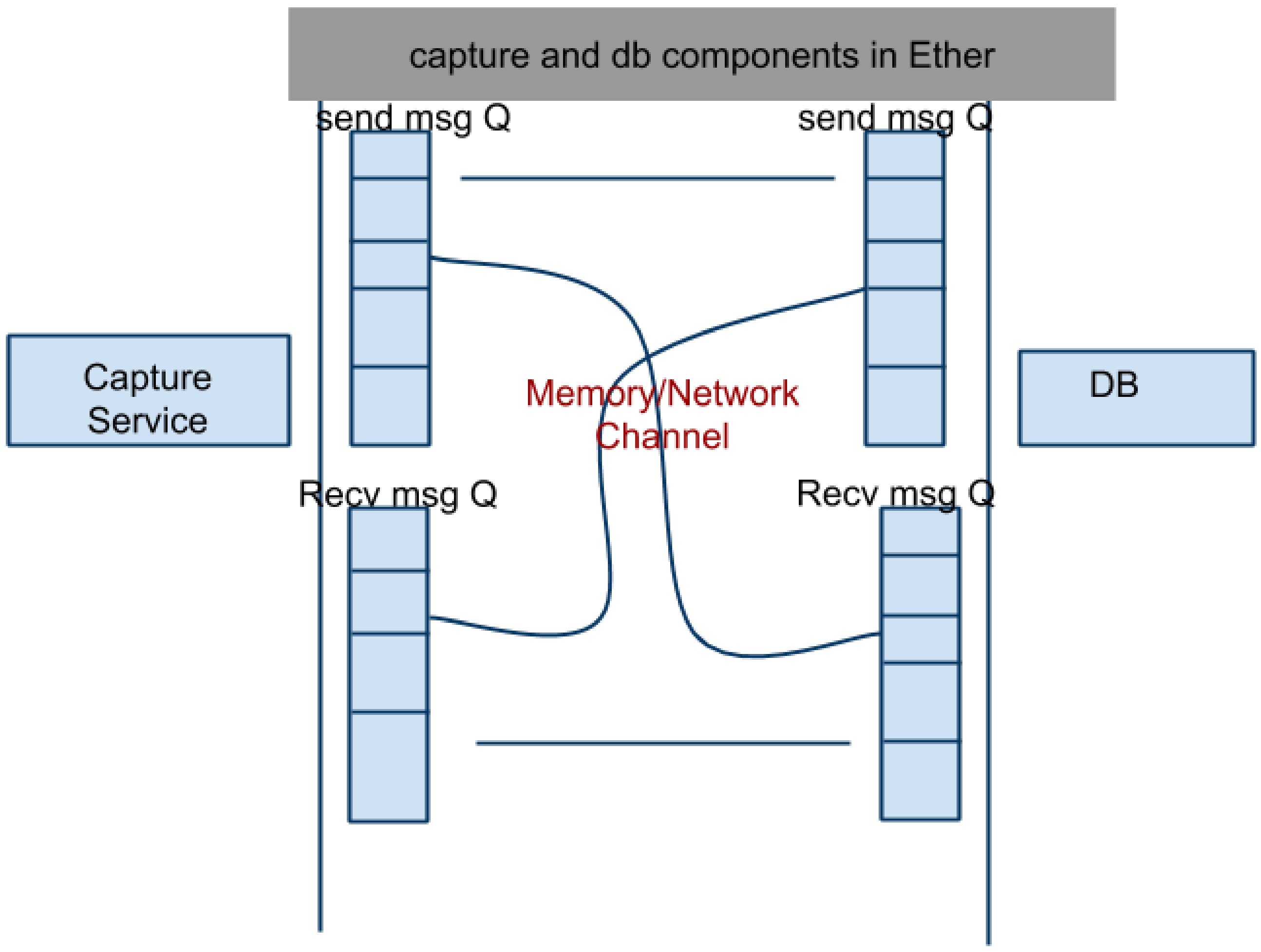
- Ether should provide a simple interface to galore components to communicate.
- components need a decent buffering mechanism while communicating.
- components need a short message system, where a SMS could be a one way quick message or a two way quick messages
- location independent messaging system. components should be totally ignorant of the location of the other components.
- support for TCP/IP, in-memory and named pipe modes of communication
- support for synchronous and asynchronous communication
- Ether is expected to provide lock-free data structures(to the possible extent) for high speed communication
- should support message aggregation or possible piggybacking; meaning the multiple messages should be packed into a single message and send/receive across
- Ether should provide interface for remote connection handling; Every component should be able to listen to a port and receive remote connections.
- should also support message forwarding.
- the message communication should be state-less. A component cannot assume anything about its target component state.
- Ether should be implemented in c++, java and python(may be for testing)

Requirements(Performance/Stability/Scalability)

- Memory Usage; ether should always use below the pre-configured memory limit
- Graceful handling of buffer overflow
- Minimum waiting time when a communication between 6 thread producers with 6 thread receivers
- Communication between the components in the same address space should always be a minimum of 10 to 20 times faster than the remote communication
- connection overhead should be minimum in case of SMS
- scalable linearly up to 6-8 producer/consumer threads

- increasing in the memory buffer size should obviously result in performance.

Picture is worth thousand Messages



Ether sub-components

we can divide Ether module into three sub-components.

- Messages
- Message Queues
- Local/Remote Connection Handlers
- Channels
- photon
- Threads
- SMS

Messages

Messages are the smallest unit of work that can be transferred between any two galore components. A message consists of a header and payload. (Ref to crp.zargo for more details). Messages can be identified with MessageType, which is useful in interpreting the message. Multiple messages can be grouped into a single message. The message group can be identified by setting the high order bits of the message type.

Message Queues
Message Queues provide the basic buffering for the data transfer between the components. The only way the components interact with each other is through the messages. Every component has two message queues; one for sending (sendQ) and one for receiving (recvQ) messages.

Local and Remote Connection Handlers
Ether provides interface for the components to be able to receive any incoming local/remote connection. This is in a way is the receiving end of the initial connection establishment. All components should always be waiting for incoming local/remote connections. This Ether sub-component is expected to provide the implementation support for the same. At the end of connection establishment, a photon object is returned. (More about photon in the next section)
One more important thing that the local/remote connection handlers do is reusing the connections. If a previous connection between capture service and database exists, it tries to reuse the connection. Ideally, the connection handlers need to reuse the previously created photon objects.

Photon
The photon object contains all the necessary data and means for further communication between the components.
For example, a typical photon object consists

- send/receive functions
- Send/Receive Message Queues

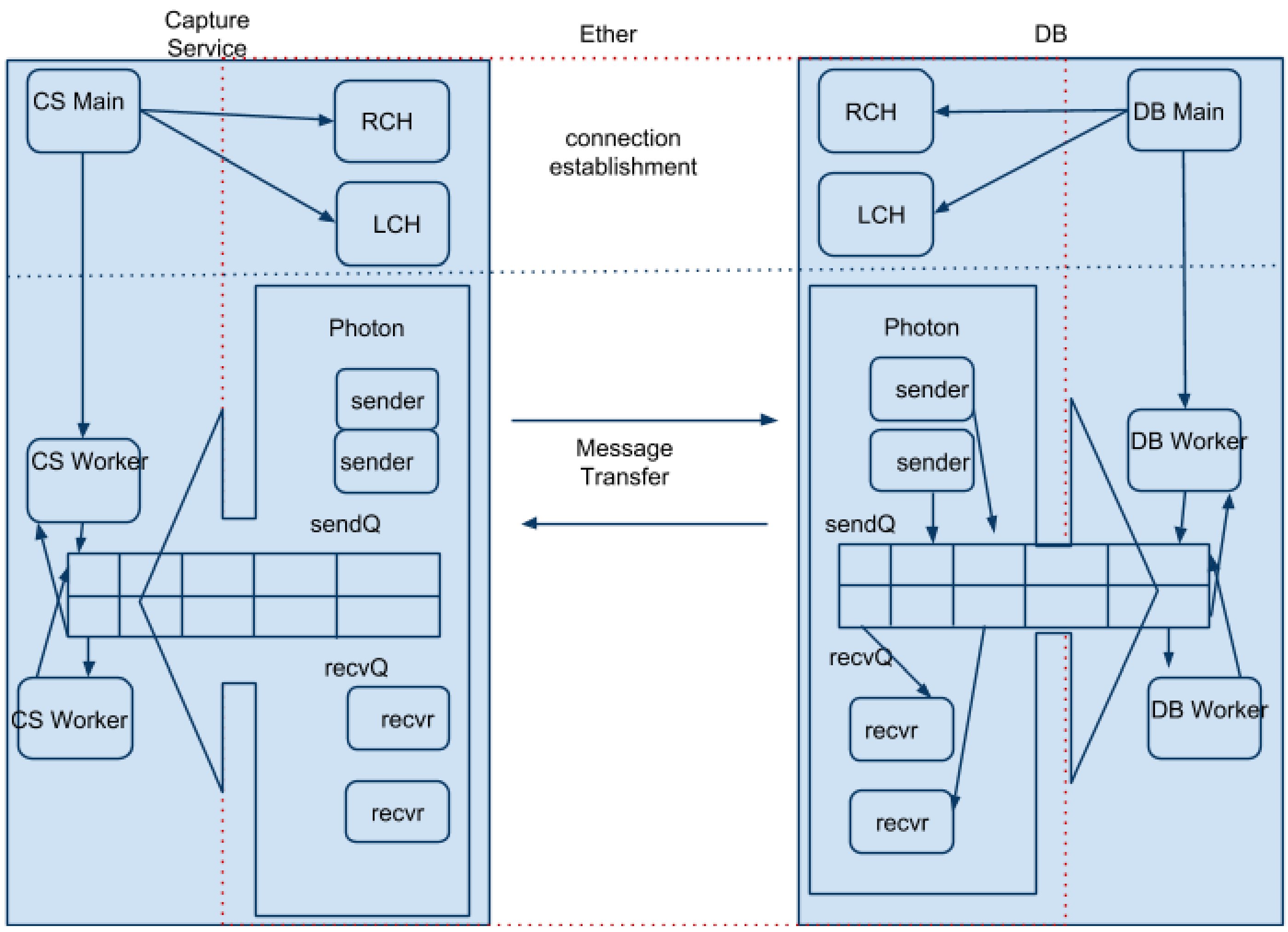
All the further communication between the components happens through the photon object. The best part of the photon object is that it can be shared by multiple producer components as well as multiple consumer components.
photon optionally can create its own sender/receiver threads (pre-configured), these threads look at the send/receive queues and do the appropriate send/receive.

Channels
Channel is the generic interface for the send/receive/connect functions. Channel hides the implementation details of the inherent medium(like, memory, network). Channel also provide interface for local/remote connection handlers for connection.

SMS
Short Message Service should be provided by photon. This should be completely stateless, we

can make it stateful by sending a session-id etc., Also, it can also be a two way sms too.

Threading Model in Ether



*all rounded rectangles in the above picture indicates a thread. please note that the photon itself is an object, but it can have optional sender/receiver threads.

How a Component interacts with Ether?

- Like the mysterious medium Ether with which the universe is filled, all the components in Galore are always are driven by Ether. Any two components registered to Ether can always communicate. A component is always reachable as long as it is registered in Ether.
- Every component should have its main thread
- the main thread instantiates the Local and Remote Connection Handlers, these can typically be threads waiting for incoming connection requests
- the main thread should also pass a callback message handler method, that deals with

different incoming message types; Ideally, the message handler callback method should be same for local and remote connection handlers

- After the connection is established(local/remote), a photon object is created, which handles the further communication.

End to End Flow

Here is how a typical message passing between capture and database can happen.

- capture service(process) at startup creates capture service main thread(CSMain); Ideally all the processes in galore should create a main thread, it is the main thread that does all the necessary. This has an advantage that, each process can be made as a thread and vice versa. The main thread registers itself to ether. As part of registration, the ether creates two threads one for local connection handler(LCH) and one for remote connection handler(RCH)
- same as the above for DB Service, DBMain thread is created first and then RCH, LCH while registration to Ether
- CSMain starts CSWorker thread to do all the work. Either of CSMain thread or CSWorker thread can start a connection to DB by using the generic connection handler(which in turn uses either local/remote connection handler to connect). If the LCH/RCH finds a connection in the pool, it immediately returns the corresponding **photon** object.
- DB-LCH/DB-RCH receives the connection(depending on whether DB service is in the same process or not), completes the connection request and returns a new **photon** object to DBMain Thread and also returns acknowledgment to the CS-LCH/CS-RCH. Based on the message type, the DBMain creates a DBWorker thread(s) and assigns the photon object to it.
- CS-LCH/CS-RCH receives the acknowledgment and completes establishing connection, returns the new **photon** object; here on the communication happens through the photon object.
- CSWorker thread starts adding the objects(PackUnpacker) to the send message queue of **photon** object. Here, if the send is synchronized, the send happens in place. However, if it is not, the photon can have its own sender/receiver threads to pick messages and do a async send/receive.
- The DBWorker thread is waiting on the **photon** receive message queue if it has any incoming messages that it can act on. If it does, it processes the incoming message. If the DBWorker thread wants to send data back to CS, it adds the message to **photon** send message queue, which either will be picked by photon sender thread or can be sent in place.
- Message Queues is the mechanism to buffer the data in between. The Queues should help us deal with scenarios like, faster producer, slower consumers or the other way. At any point of time, the size of the message queue is nothing but the size of the ether

buffer for that connection.