Nicolò Rivetti

Concurrent and Parallel Programming Project

Total Order Broadcast

Project Description

This project aims to implement, using **C/MPI**, the **Total Order Broadcast** abstraction presented in the book “Introduction to Reliable and Secure Distributed Programming” by Cachin, Guerraoui and Rodrigues [1].

The abstraction is built through the following **protocol stack**:

- **Consensus Based Total Order Broadcast**
- **Flooding Consensus**
- **Eager Reliable Broadcast**
- **Best Effort Broadcast**
- **Perfect Failure Detector**
- **Perfect Point to Point Link**

Building Blocks

The algorithms are based on a **asynchronous** **event based composition model**, hence each layer is modeled as components which interact through events generation and consumption.

- **Event Queue & Runtime**

The protocol layers **register events** to the first, while the latter **consumes events** causing the related handlers to be executed.

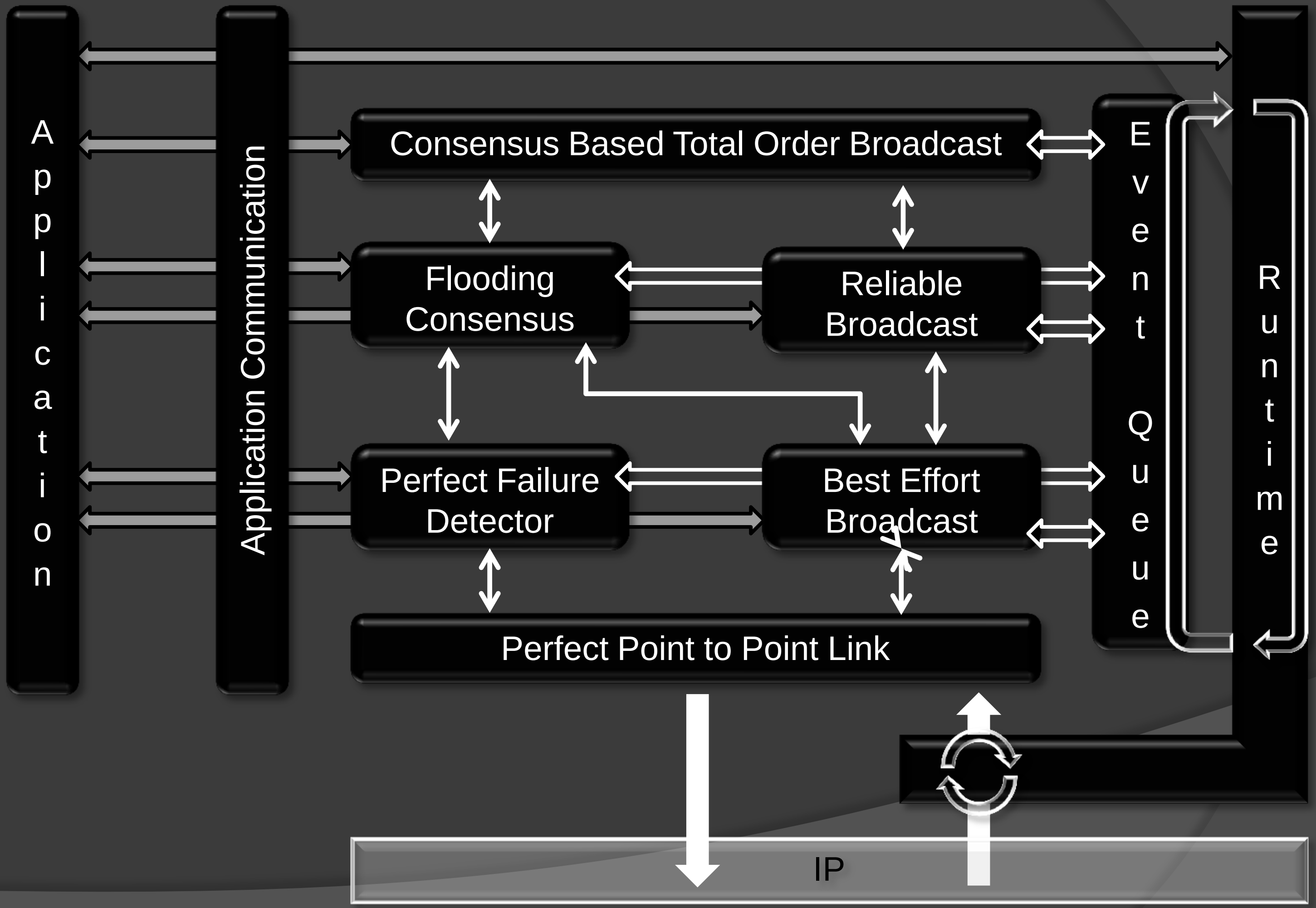
- **Perfect P2P Link & Runtime**

MPI provides a point to point communication primitive fitting the PP2P Link properties, however the messages reception is not completely asynchronous. To add this capability, the runtime takes care of checking for new messages and post them as event to the relevant component.

- **Application Communication**

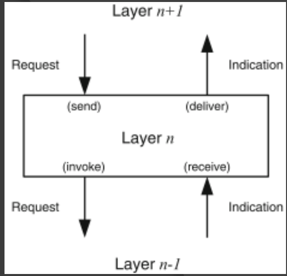
To allow maximal freedom to the application layer, the library lives in it’s own process, hence this module hides the communication details between the runtime and the application process.

Overall Architecture



Event Queue

Using a event based composition model, all module share a common structure which implements the **Indication / Request** API shown in [1].



- Each module allows upper layers to **register a callback** which is a function pointer to a procedure that will be called as **Indication**.
- Each module expose a procedure to be called as **Request**.
- The Request and Indication procedures implementation will only **register an event** to the Event Queue, as a function pointer, a pointer to data and the data size

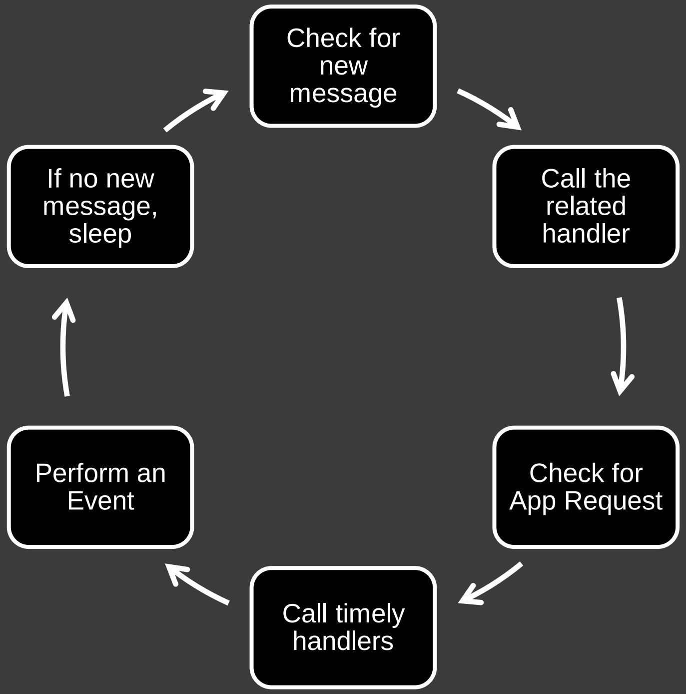
```
typedef struct event_struct {
    int (*callback)(int * data, int size);
    void * data;
    int size;
    struct event_struct *next;
} event_t;
```

```
int _perform_event() {

    // There are no events to perform
    if (event_head->next == NULL ) {
        return 0;
    }

    sprintf(log_buffer, "RUNTIME INFO: Performing Event (Events in queue:%d).", event_num);
    runtime_log(DEBUG_EVENTQUEUE);

    event_t * event = event_head->next;

    // Performing and removing the event
    int ret = event->callback(event->data, event->size);

    // If it's the last event in the queue, event_tail must point to the head
    if (event_tail == event_head->next) {
        event_tail = event_head;
        event_head->next = NULL;
    } else {
        event_head->next = event->next;
    }

    // Deallocating event_struct, pointed data should be handled by callback
    free(event);

    event_num--;

    return ret;
}
```

```
// Abstraction Logic Implementation
int _best_effort_broadcast_send(int *message, int count, int handle);
int _best_effort_broadcast_send_callback(int *data, int size);
int _best_effort_broadcast_receive(int *message, int count, int sender);

// Commodity and Utility Functions
int _best_effort_broadcast_register_callback(int (*callback_func)(int*, int, int), int* handle);
```

Runtime

The Runtime is the heart of the system. After setting up all modules required by the application (using flags), it loops performing the following tasks:



- Through the MPI_Iprobe function, the runtime checks if a new messages is ready to be received.

- If it’s the case, then it actually receives the message. Given the TAG of the message, the callback registered whit it is called

- Checks if the application layer has sent some messages through the App Comm Module

- Execute all the callback registered as timely.

- Consume one event registered on the Queue Event

- If no messages has been received, it sleeps for few ms

Serial Number Management (1/2)

Many protocols are based on the assumption that each message is tagged with a **unique identifier**, which could be the pair <senderId, Serial Number>. Eventually, since **memory is limited**, the Serial Number (SN) will **overflow** and must be set back to 0, **breaking the uniqueness assumption**.

In this project two way of dealing with this issue have been devised, the first is used to manage the *Uniform Consensus* Instances SN

```c
/*
 * Commodity function to retrieve the instance struct (if it exists) given it's id
 */
instance * getinstance(int id) {
    instance *parent = head;

    gettimeofday(&gettimeofdayArg, NULL );
    unsigned long currentTS = gettimeofdayArg.tv_sec;

    while (parent->next != NULL ) {
        if ((parent->next->decision != NULL )&&(parent->next->decidedTs < currentTS - MIN_DELAY)){
        instance *ist = parent->next;
```

```c
        // Delete Record

        parent->next = parent->next->next;
        free(ist);
    } else {

        if (parent->next->instance_id == id) {
            return parent->next;
        }
        parent = parent->next;
    }
}

    return NULL ;
}
```

```c
struct instance_struct {
    int instance_id;
    int callback;
    int upper_layer_id;
    int round;
    proposal **proposals;
    proposal *decision;
    unsigned long decidedTs;
    int **receivedFrom;
    struct instance_struct *next;
};

typedef struct instance_struct instance;
static instance* head;
```

- A new SN is accepted only if it has not been recorded in the SN data structure.
- When a SN is stored, the record contains also a timestamp (TS).
- Each time the data structure is looked up, if record TS < current TS - Delay, then the record is deleted

Serial Number Management (2/2)

This mechanism is used to manage the arriving messages for *Eager Reliable Broadcast* and *Consensus Based Total Order Broadcast*

The second solution is **less memory consuming** but a bit more complex.
The available space is divided in quarters , the four values are stored in a data structure as G0,G1, G2, G3 (Guards) and have a TS associated. T is the next expected SN.

```c
struct serialNumber_struct {
    unsigned int treshold;
    unsigned int max;
    unsigned int guard;
    unsigned int guards[4];
    long timestamps[4];
};
```

```c
#define MAX_UINT ((unsigned int ) ~0)
#define QUARTER0 ((unsigned int ) 0)
#define QUARTER1 ((unsigned int ) (MAX_UINT - 1) / 4)
#define QUARTER2 ((unsigned int ) QUARTER1 * 2 + 1)
#define QUARTER3 ((unsigned int ) QUARTER1 * 3 + 2)
```

```c
int incrementNextSerialNumber(serialNumber *sn, unsigned int id, long ts) {
    if (sn->treshold < sn->guards[sn->guard]) {
        if (id >= sn->treshold && id < sn->guards[sn->guard]) {
            if (id == sn->treshold) {
                incrHighTreshold(sn);
                incrLowTreshold(sn, ts);
            }
            return 0;
        }
    } else if (sn->treshold > sn->guards[sn->guard]) {
        if ((id >= sn->treshold && id <= sn->max) || (id >= 0 && id < sn->guards[sn->guard])) {
            if (id == sn->treshold) {
                incrHighTreshold(sn);
                incrLowTreshold(sn, ts);
            }
            return 0;
        }
    }
    printf("ERROR! threshold: %u, id: %u, ts: %u, guard:%u, guardValue: %u\n", sn->treshold, id, ts, sn->guard,
        sn->guards[sn->guard]);
    return -1;
}
```

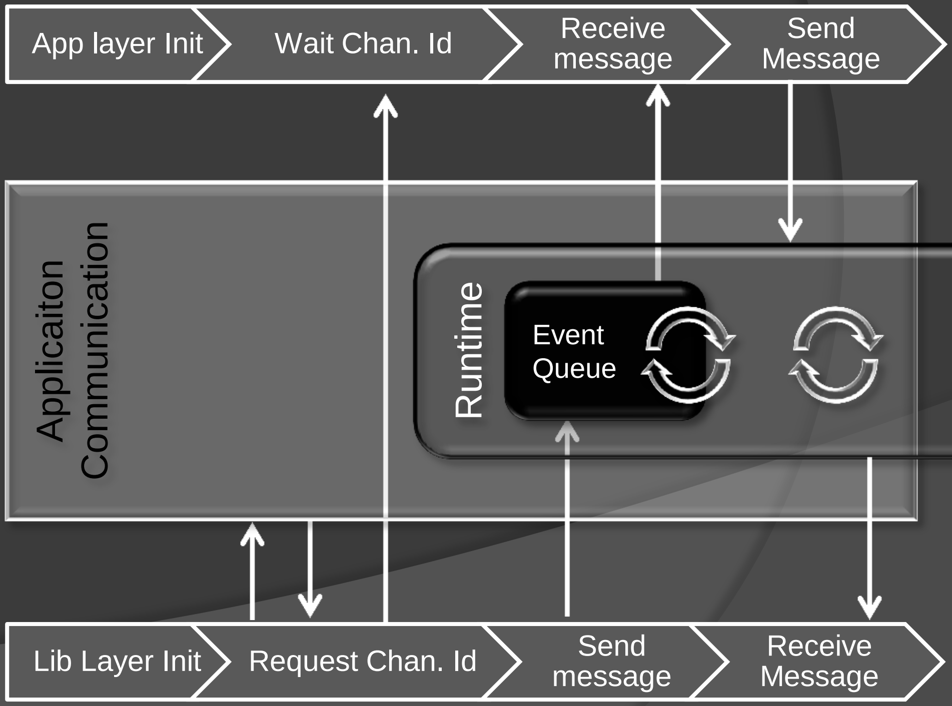
- A new SN is accepted only if $T \leq SN < G$, and has not been recorded in the SN data structure.
- When a SN is stored, if SN = T, then T is increased. If SN > current G and TS > current G TS, then G is moved to the next Guard.
- Each time the data structure is looked up, all SN = T are deleted (T is increased after each deletion)

Application Communication

Only one instance of MPI can be created per process. To **avoid unwanted interaction** between MPI API calls either the application layer must not use MPI or the runtime must live in **a new process**.

The latter was considered the best approach, requiring a module managing inter process communication

- Each module can create a channel through the App Comm module with the Application layer.
- The communication is done using **message queues**. Channels are implemented with **messages tag**.
- The initialization function of a module will ask the App Comm module for a free channel Id.
- The two parties of each module, given the channel Id can use the App Comm API to communicate



Perfect Failure Detector

- PFD1: Strong completeness: Eventually, every process that crashes is permanently detected by every correct process.
- PFD2: Strong accuracy: If a process p is detected by any process, then p has crashed.

The Perfect failure detector requires is made of two main functions. One is registered as timely callback and the other as receive callback to the runtime:

- One check if all currently correct process have replied to the Heartbeat Request and trigger a crash event if not, calling the handler registered from the other layers and flagging the process as crashed.
- The other is triggered by received messages, replies to Heartbeat Request orregisters Heartbeat Replies.

upon event $\langle$ *Timeout* $\rangle$ **do**
    forall $p \in \Pi$ **do**
        if $(p \notin alive) \wedge (p \notin detected)$ **then**
            $detected := detected \cup \{p\}$;
            trigger $\langle \mathcal{P}, Crash \mid p \rangle$;
        trigger $\langle pl, Send \mid p, [\text{HEARTBEATREQUEST}] \rangle$;
    $alive := \emptyset$;
    $starttimer(\Delta)$;

upon event $\langle pl, Deliver \mid q, [\text{HEARTBEATREQUEST}] \rangle$ **do**
    trigger $\langle pl, Send \mid q, [\text{HEARTBEATREPLY}] \rangle$;

upon event $\langle pl, Deliver \mid p, [\text{HEARTBEATREPLY}] \rangle$ **do**
    $alive := alive \cup \{p\}$;

```c
        // Check for each process
        for (i = 0; i < num_procs; i++) {

            if ((detected[i] == 0) && (alive[i] == 0)) {
                // The process has not replied to the heartbeat request and was not already detected,
                // hence the process is flagged as failed and is added to the process crashed in this
                // period
                sprintf(log_buffer, "FD INFO: Failure of process %d", i);
                runtime_log(INFO_PERFECTFAILUREDETECTOR);
                detected[i] = 1;
                detected_num++;
                currently_detected[currently_detected_num] = i;
                currently_detected_num++;
            } else if (detected[i] == 0) {
                // If the process was not detected and has replied to the heartbeat request,
                // then the process is alive and a new heartbeat request is issued
                sprintf(log_buffer, "FD INFO: Process %d alive", i);
                runtime_log(INFO_PERFECTFAILUREDETECTOR);
#ifdef BUFFEREDSEND
                MPI_Bsend(&heartbeat_request_message, 1, MPI_INT, i, runtime_receive_handle, comm);
#endif
```

The buffered mode send (MPI_Bsend) is used in this module since a fixed size message is exchange. Hence, given the number of processes it is easy to compute a upper bound on the buffer size to be attached in order to allow the buffered communication

Best Effort Broadcast

- **BEB1: Validity: If a correct process broadcasts a message m , then every correct process eventually delivers m.**
- **BEB2: No duplication: No message is delivered more than once.**
- **BEB3: No creation: If a process delivers a message m with sender s , then m was previously broadcast by process s.**

- When a messages is sent, the module sends along the id assigned to the upper layer which requested the send.
- The receive function registered to the runtime read the id contained in the header of the messages and call the callback related to this id that was registered by the upper layer.

upon event ⟨ *beb*, *Broadcast* | m ⟩ **do**
    forall $q \in \Pi$ **do**
        trigger ⟨ *pl*, *Send* | q, m ⟩;

upon event ⟨ *pl*, *Deliver* | p, m ⟩ **do**
    trigger ⟨ *beb*, *Deliver* | p, m ⟩;

```
#ifndef BUFFEREDSEND
    if (_remove_buffer() == -1)
        return -1;

    MPI_Request** mpi_requests = (MPI_Request**) malloc(sizeof(MPI_Request*) * num_procs);
    if (mpi_requests == NULL ) {
        sprintf(log_buffer, "BEB ERROR:Cannot allocate the mpi requests pointer table.");
        runtime_log(ERROR_BEBROADCAST);
        return -1;
    }
    if (_add_buffer(message, mpi_requests) == -1)
        return -1;

#endif

    for (current_process = 0; current_process < num_procs; current_process++) {
        if (detected[current_process] == 0) {
#ifndef BUFFEREDSEND
            MPI_Isend(message, size, MPI_INT, current_process, runtime_receive_handle, comm,
                    mpi_requests[current_process]);
#endif
```

```
    while (parent->next != NULL ) {
        buffer_data * bd = parent->next;
        flag = 0;
        for (i = 0; i < num_procs; i++) {
            if (detected[i] == 0) {
                MPI_Test(bd->mpi_requests[i], &flag, &status);
                if (!flag || (status.MPI_SOURCE == MPI_ANY_SOURCE && status.MPI_TAG == MPI_ANY_TAG)) {
                    flag = 0;
```

The function that handles send requests uses the asynchronous semantic (MPI_Isend) instead of the buffered mode since it is hard to predict the size of the buffer to be attached. However this requires to manage the MPI_Request stuct for each send. Each send is recorded in a data structure. When this is visited, the module checks if the relates send is terminated, if it has it deletes the record.

Reliable Broadcast

- **RB1: Validity: If a correct process p broadcasts a message m , then p eventually delivers m .**
- **RB2: No duplication: BEB2**
- **RB3: No creation: BEB 1**
- **RB4: Agreement: If a message m is delivered by some correct process, then m is eventually delivered by every correct process.**

```c
int add_to_delivered(unsigned int id, int real_sender) {
    // Delivered (removed ids)
    if (isAllowedSerialNumber(proc_data_set[real_sender]->gc_threshold, id) == -1) {
        return -1;
    }

    delivered * father = proc_data_set[real_sender]->delivered_set;

    // Skips messages with id < thresold, until the list ends or the message with id = 0 is found
    while (father->next != NULL && isLowerSerialNumber(proc_data_set[real_sender]->gc_threshold, father->next->id) == 0) {
        father->next = father->next->next;
    }

    // Removes oldest contiguous messages from delivered set
    while (father->next != NULL && isNextSerialNumber(proc_data_set[real_sender]->gc_threshold, father->next->id) == 0) {
        delivered * d = father->next;

        gettimeofday(&gettimeofdayArg, NULL );
        incrementNextSerialNumber(proc_data_set[real_sender]->gc_threshold, father->next->id, gettimeofdayArg.tv_sec);

        father->next = father->next->next;
        free(d);
    }
```

The Reliable Broadcast implements two functions:

- The send function uses the BEB layer to send a broadcast, the init has taken care to get the id used by BEB to identify messages for this module

upon event ⟨ *rb*, *Broadcast* | *m* ⟩ **do**
    trigger ⟨ *beb*, *Broadcast* | [DATA, *self*, *m*] ⟩;

upon event ⟨ *beb*, *Deliver* | *p*, [DATA, *s*, *m*] ⟩ **do**
    if $m \notin delivered$ **then**
        $delivered := delivered \cup \{m\}$;
        trigger ⟨ *rb*, *Deliver* | *s*, *m* ⟩;
        trigger ⟨ *beb*, *Broadcast* | [DATA, *s*, *m*] ⟩;

- A function which is registered to the BEB layer is called when a message for this module has been received. This checks if the received messages has already been delivered through a mechanism that uses the second mean of managing Serial Number.

```c
int _eager_reliable_broadcast_send(int *message, int count, int handle) {
    sprintf(log_buffer, "EAGER RELIABLE BROADCAST INFO: Registering message \"%d\"
    runtime_log(DEBUG_EAGERRELIABLEBROADCAST);

    // Copying the message, adding EAGER RELIABLE BROADCAST callback.
    int * buffer = (int*) malloc(sizeof(int) * (count + 3));
    buffer[0] = handle;
    buffer[1] = serial_number++;
    buffer[2] = my_rank;
    copy_to_array(&buffer[3], message, count);
    _best_effort_broadcast_send(buffer, count + 3, best_effort_broadcast_handle);
    free(buffer);
    return 0;
}
```

Consensus

- C1: Termination: Every correct process eventually decides some value.
- C2: Validity: If a process decides v , then v was proposed by some process.
- C3: Integrity: No process decides twice.
- C4: Agreement: No two correct processes decide differently..

The Consensus X functions:

- The propose function gets an array of values, sorts it and delete duplicates before sending it through the BEB layer along with the id given by the calling layer and the id wrt the module.

- A function is registered to the BEB layer and when it is called reads the consensus id and either retrieves the instances data or creates a new instances with the id.

- When a propose or a crash event is received, a function is called which checks if the decide conditions have been reached, retrieves the latest proposed set, sends it as decided set through the BEB layer and triggers the decided event calling the callback registered by the layer which requested the propose.

```
upon event ⟨ P, Crash | p ⟩ do
    correct := correct \ {p};

upon event ⟨ c, Propose | v ⟩ do
    proposals[1] := proposals[1] ∪ {v};
    trigger ⟨ beb, Broadcast | [PROPOSAL, 1, proposals[1]] ⟩;

upon event ⟨ beb, Deliver | p, [PROPOSAL, r, ps] ⟩ do
    receivedfrom[r] := receivedfrom[r] ∪ {p};
    proposals[r] := proposals[r] ∪ ps;

upon correct ⊆ receivedfrom[round] ∧ decision = ⊥ do
    if receivedfrom[round] = receivedfrom[round − 1] then
        decision := min(proposals[round]);
        trigger ⟨ beb, Broadcast | [DECIDED, decision] ⟩;
        trigger ⟨ c, Decide | decision ⟩;
    else
        round := round + 1;
        trigger ⟨ beb, Broadcast | [PROPOSAL, round, proposals[round − 1]] ⟩;

upon event ⟨ beb, Deliver | p, [DECIDED, v] ⟩ such that p ∈ correct ∧ decision = ⊥ do
    decision := v;
    trigger ⟨ beb, Broadcast | [DECIDED, decision] ⟩;
    trigger ⟨ c, Decide | decision ⟩;
```

The managing of the consensus id is done though the first mechanism explained earlier

Total Order Broadcast

- TOB1: Validity: If a correct process p broadcasts a message m, then p eventually delivers m .
- TOB2: No duplication: No message is delivered more than once.
- TOB3: No creation: If a process delivers a message m with sender s, then m was previously broadcast by process s.
- TOB4: Agreement: If a message m is delivered by some correct process, then m is eventually delivered by every correct process.
- TOB5: Total order: Let m 1 and m 2 be any two messages and suppose p and q are any two correct processes that deliver m1 and m2. If p delivers m 1 before m 2, then q delivers m 1 before m 2.

The message id in the delivered data structure are managed as in the reliable module

The Total Order Broadcast module registers handlers both at the Reliable broadcast layer and at the Consensus layer.

- When the send function is called, the module sends the message trough RB layer.
- When the function registered to the RB layer is called, the given message is stored in the unordered data structure.
- If unordered is not empty and no consensus instances are running, a consensus instances with the id (wrt TOB) of the messages in unordered is launched.
- When the consensus returns the decided set, the decided id are removed form unordered and added to delivered and then the deliver event is triggered, calling the callback registered by the calling layer.

```
upon event ⟨ tob, Broadcast | m ⟩ do
    trigger ⟨ rb, Broadcast | m ⟩;

upon event ⟨ rb, Deliver | p, m ⟩ do
    if m ∉ delivered then
        unordered := unordered ∪ {(p, m)};

upon unordered ≠ ∅ ∧ wait = FALSE do
    wait := TRUE;
    Initialize a new instance c.round of consensus;
    trigger ⟨ c.round, Propose | unordered ⟩;

upon event ⟨ c.r, Decide | decided ⟩ such that r = round do
    forall (s, m) ∈ sort(decided) do                    // by
        trigger ⟨ tob, Deliver | s, m ⟩;
    delivered := delivered ∪ decided;
    unordered := unordered \ decided;
    round := round + 1;
    wait := FALSE;
```

Bibliography

- [1] Reliable and Secure Distributed Programming (Second Edition), Christian Cachin, Rachid Guerraoui, Luis Rodrigues

- [2] Parallel Programming in C with MPI and OpenMP, Michael J. Quinn

- [3] An Introduction to Parallel Programming, Peter Pacheco

- [4] MPI: The Complete Reference http://www.netlib.org/utk/papers/mpi-book/mpi-book.html

- [5] Setting up a Beowulf Cluster Using Open MPI on Linux, http://techtinkering.com/2009/12/02/setting-up-a-beowulf-cluster-using-open-mpi-on-linux/

- [6] C Programming language (Second Edition), Brian W. Kernighan, Dennis M. Ritchie

- [7] Open MPI: Open Source High Performance Computing, http://www.open-mpi.org/