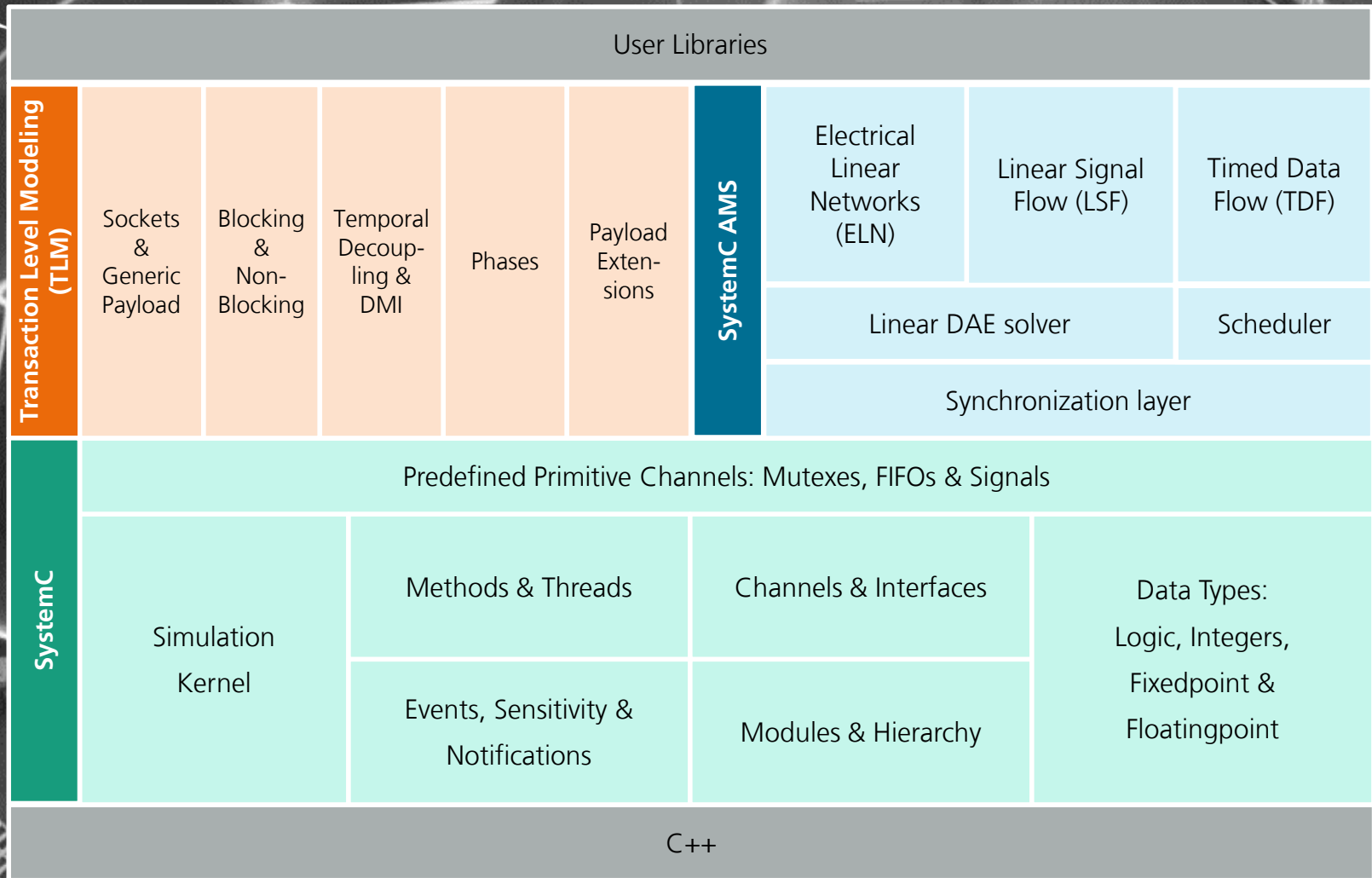


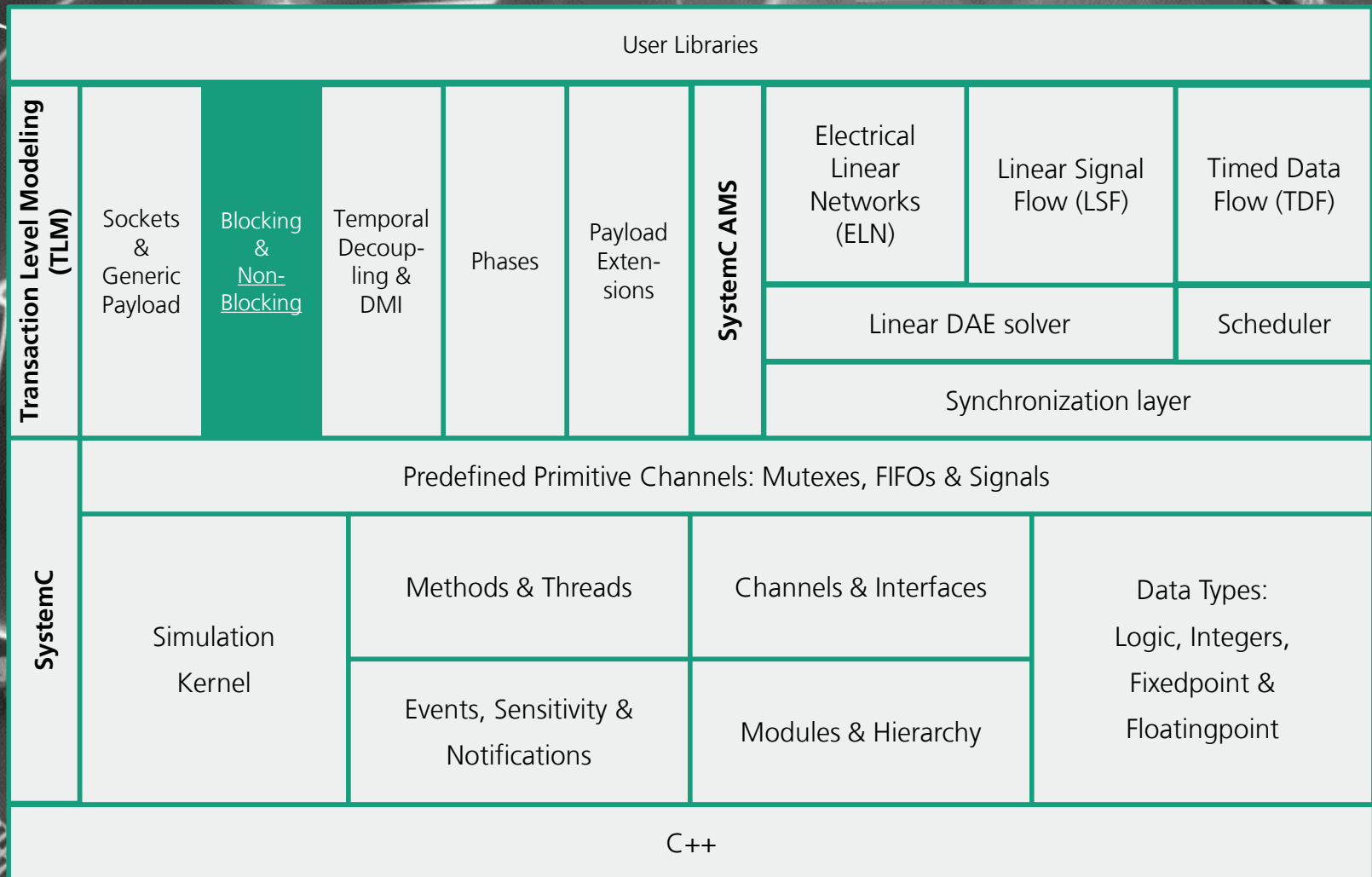
# SystemC and Virtual Prototyping

Dr. Matthias Jung, Fraunhofer Institute IESE

[matthias.jung@iese.fraunhofer.de](mailto:matthias.jung@iese.fraunhofer.de)







# Recap: TLM Coding Styles and Mechanisms

## TLM Use Cases

SW Application  
Development

SW Performance  
Analysis

Architecture  
Analysis

Hardware  
Verification

## TLM 2.0 Coding Style *(Just Guidelines)*

Loosely-Timed (LT)

Single-phase, blocking API

Multi-phase, non-blocking API

Approximately-Timed (AT)

## TLM Mechanisms *(Definitive API for enabling Interoperability)*

Blocking  
transport

DMI

Quantum  
(Keeper)

Sockets

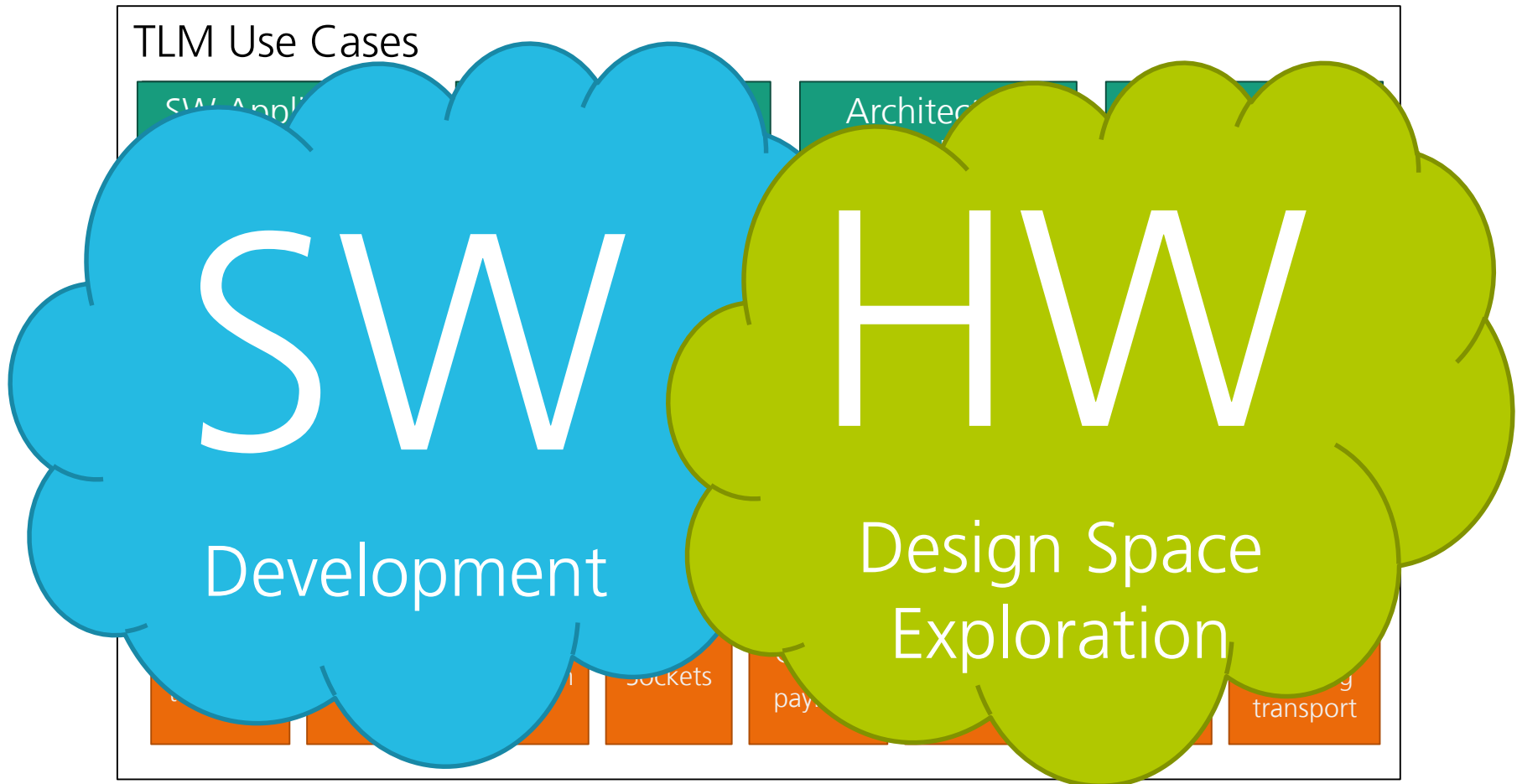
Generic  
payload

Extensions

Phases

Non-  
blocking  
transport

# Recap: TLM Coding Styles and Mechanisms



# Recap: Coding Styles in TLM

## ■ Loosely-Timed (LT):



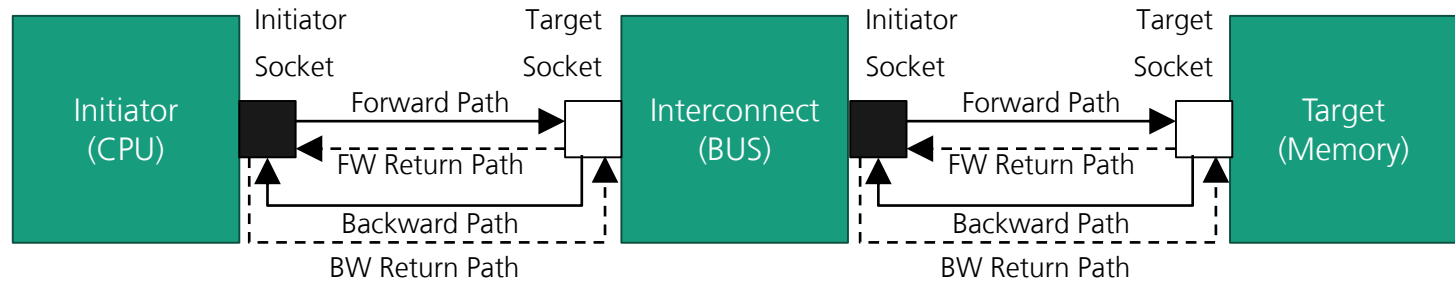
- As fast as possible
- Sufficient timing detail to boot OS and run multicore systems and to develop SW or drivers
- Processes can run ahead of simulation time (temporal decoupling)
- Each transaction completes in one blocking function call
- Usage of Direct Memory Interface (DMI) e.g. for boot process

## ■ Approximately-Timed (AT):

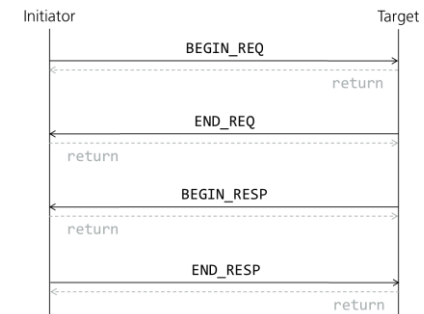


- Accurate enough for performance modelling
- Sufficient for architectural HW design space exploration
- Processes run in lockstep with simulation time
- Each transaction has usually 4 timing points i.e. 4 function calls (extensible if required, also less possible); non-blocking behavior
- More detailed than LT and therefore also slower than LT

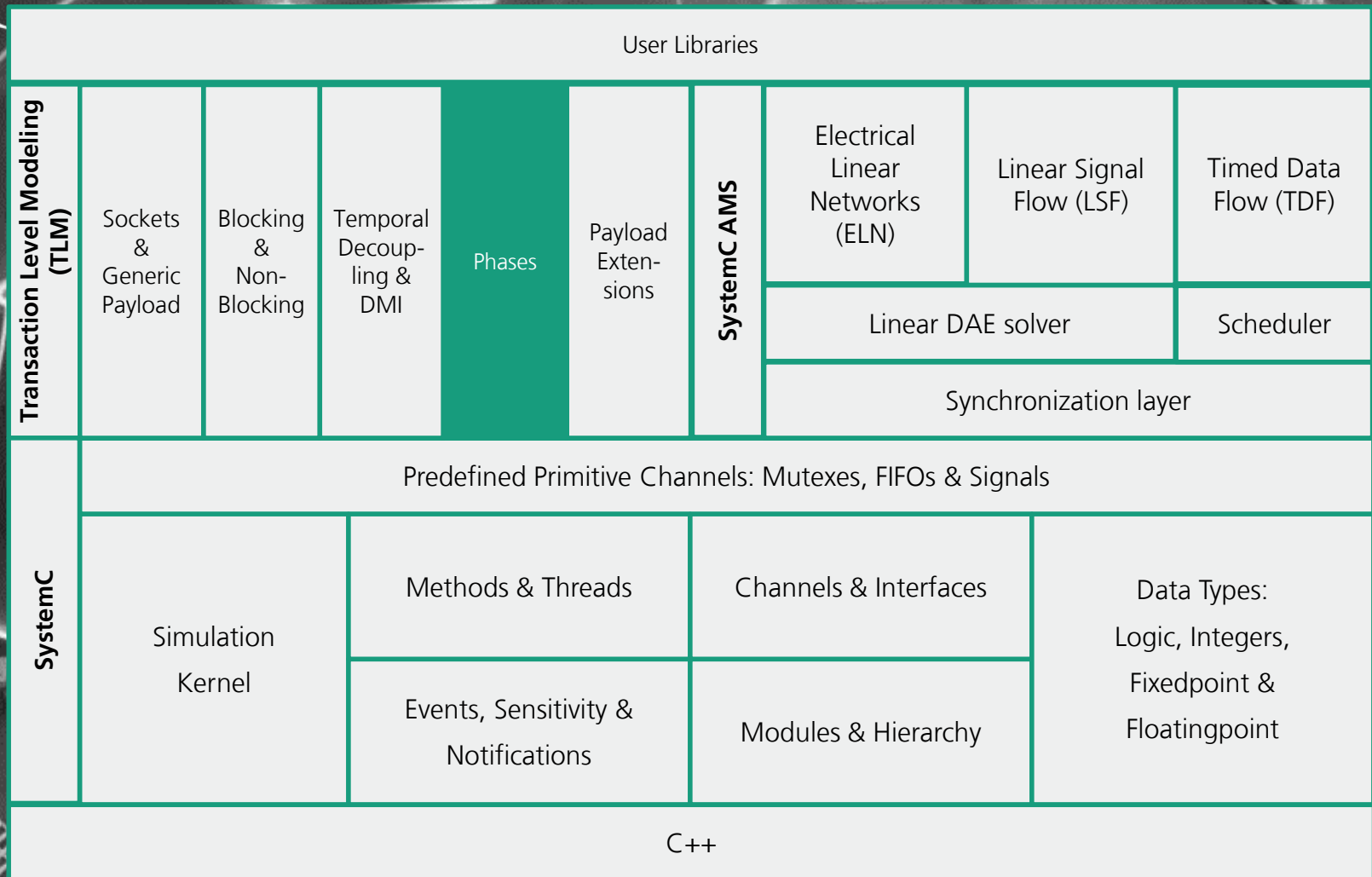
# Initiators, Targets and Interconnect



- References to the object are passed along the forward and backward paths:
  - LT uses Forward and Return Path
  - AT uses Forward, Backward, FW Return Path, BW Return Path
- AT uses non-blocking transport
- Time is handled with Payload Event Queues (PEQs)
- Allows modelling of O-O-O Cores and Backpressure
- Base Protocol

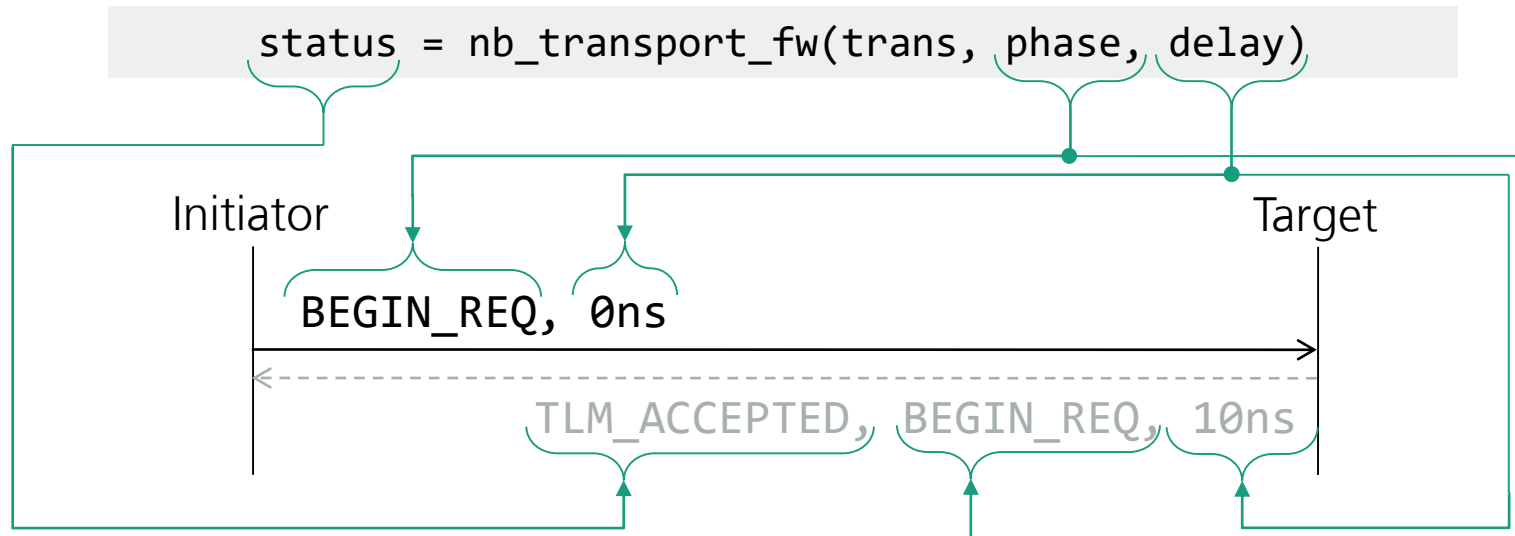




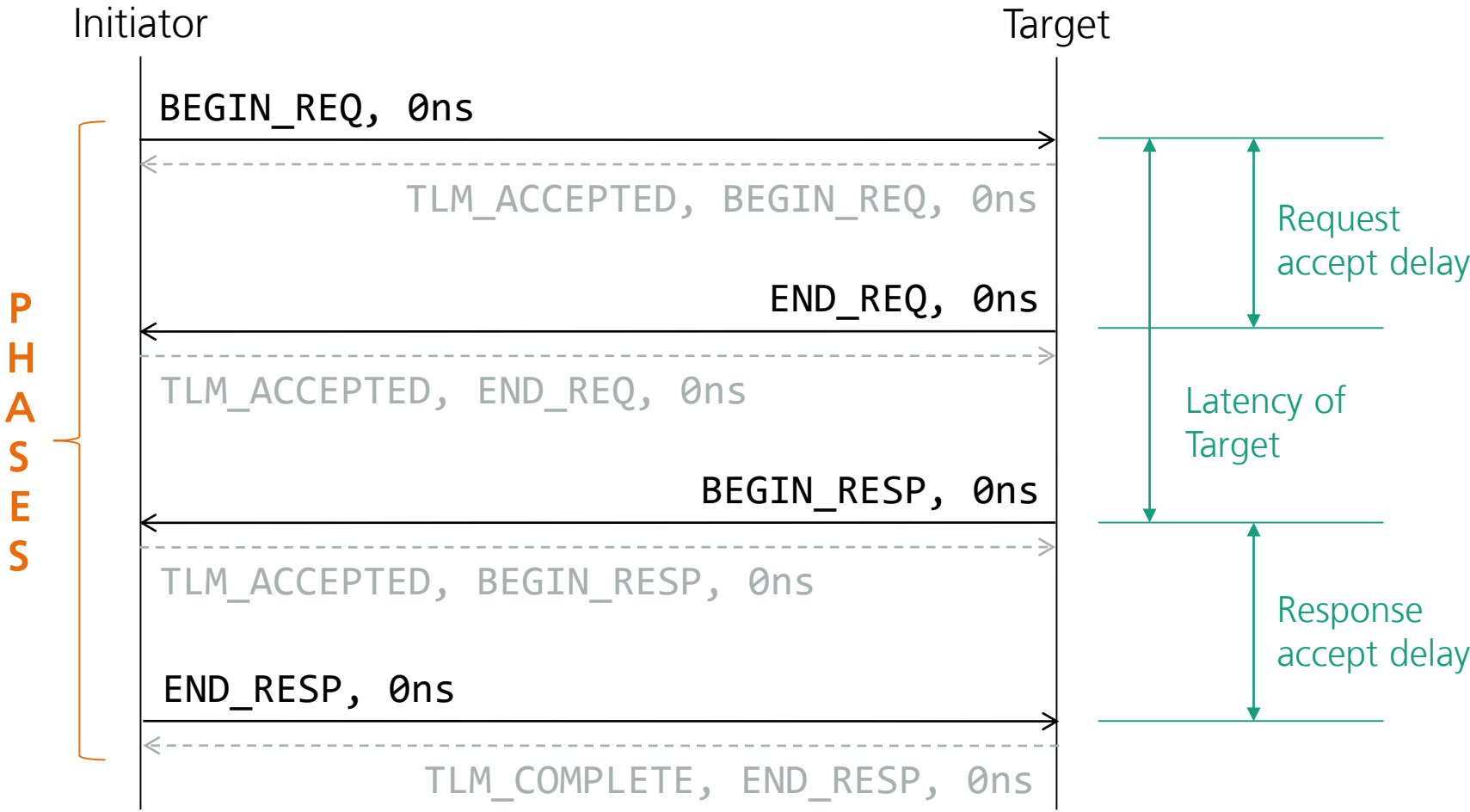




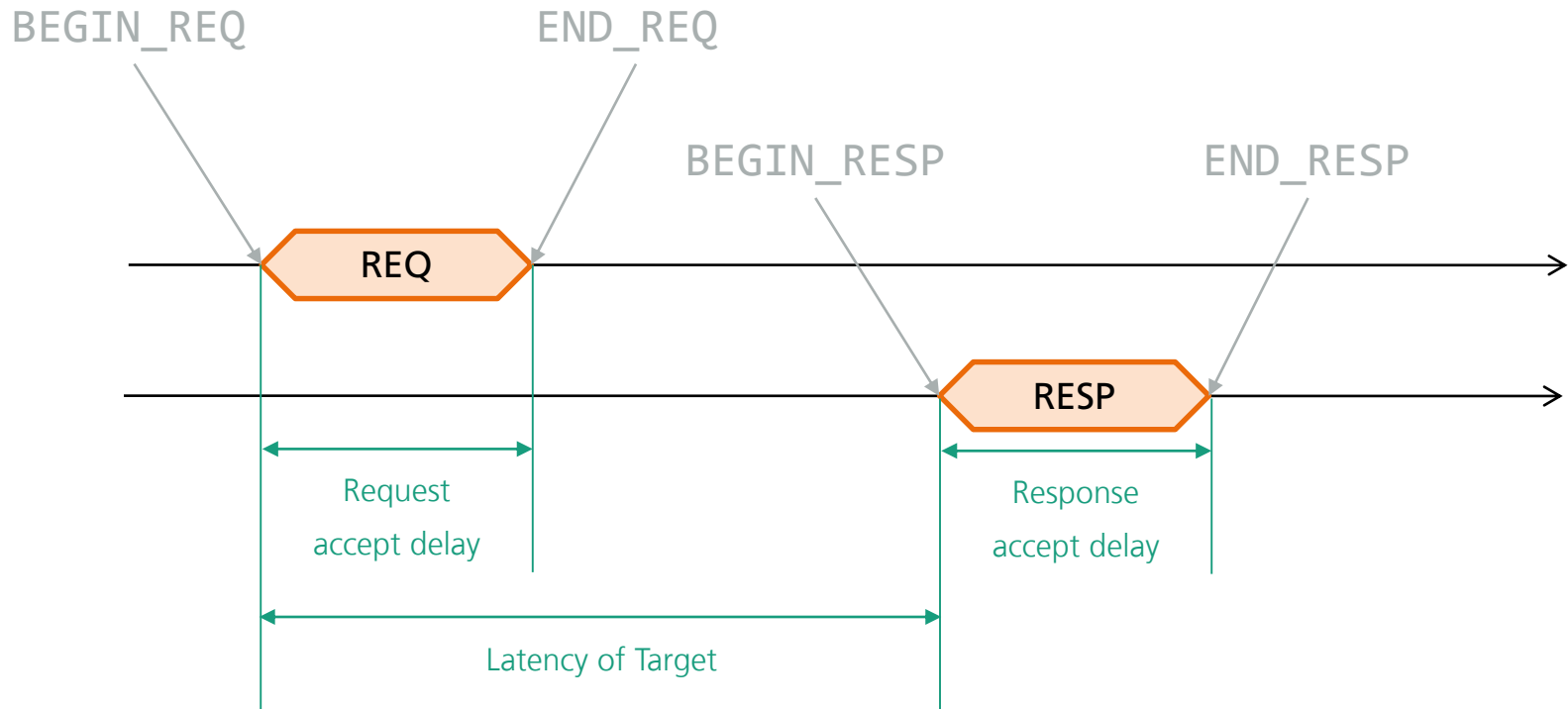
# Non-Blocking Transport (AT) Base Protocol



# Base Protocol Rules [1]: Using BW Path



# Alternative View



# Base Protocol Rules: Using BW Path

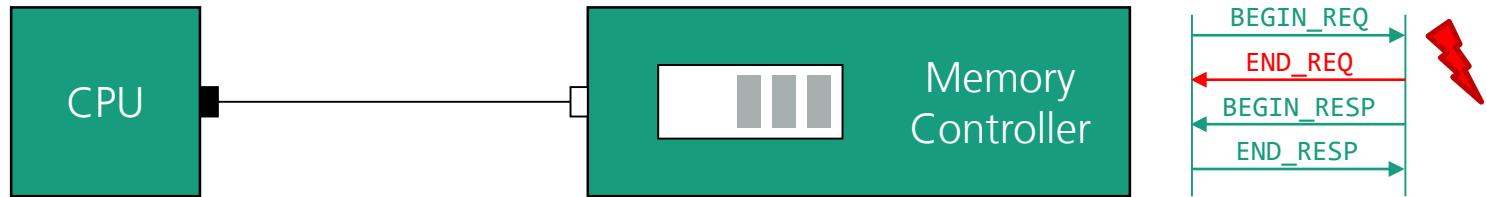
- Base Protocol phases
  - BEGIN\_REQ → END\_REQ → BEGIN\_RESP → END\_RESP
  - Must occur in increasing simulation time order
  - Phases must change with each call
- nb\_transport\_fw must not call nb\_transport\_bw directly and vice versa (PEQ!)
- Generic Payload memory management rules (See TLM Advanced)
- Extensions must be ignorable (See TLM Advanced)
- Target should handle mixed b\_transport / nb\_transport

# Base Protocol Rules: The Exclusion Rule



- Initiators and targets must honor the *Exclusion Rule*:
  - An Initiator must not send a new request (**BEGIN\_REQ**) until it has received the **END\_REQ** from the previous transaction
  - An target must not send the next **BEGIN\_RESP** until it has received the **END\_RESP** from the previous transactions
- The exclusion rules enable flow control like back pressure:
  - E.g. if an input buffer of a target is full the target can defer the sending of **END\_REQ** for the transaction that filled the buffer, until the buffer has available space again
  - An RTL ready signal can be modeled by deferring **END\_REQ**
  - However, since it is non-blocking, the target can do something else, e.g. sending

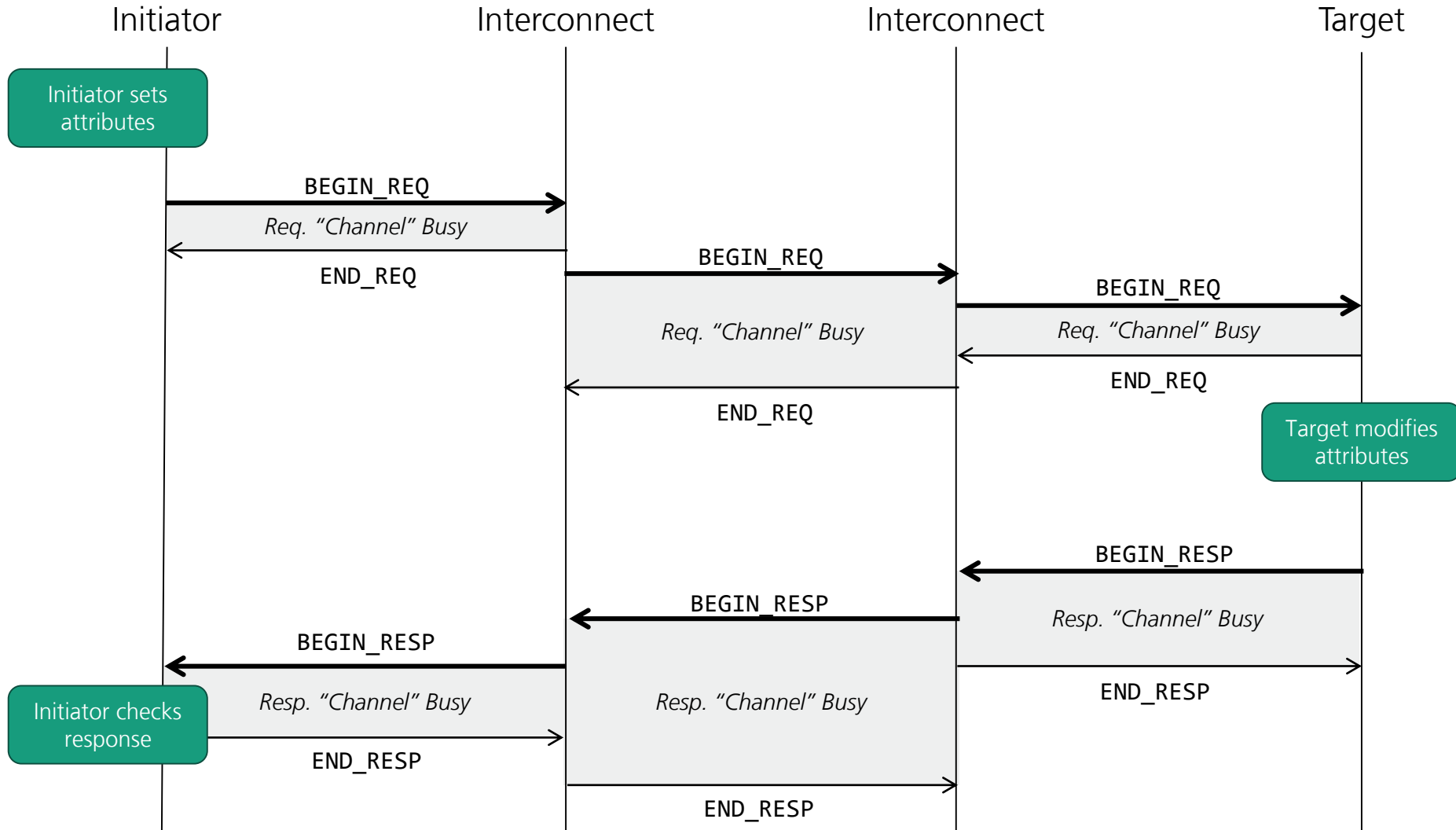
# Modelling of Backpressure



- The exclusion rules enable flow control like back pressure:
  - E.g. if an input buffer of a target is full the target can defer the sending of **END\_REQ** for the transaction that filled the buffer, until the buffer has available space again
  - An RTL ready signal can be modeled by deferring **END\_REQ**
  - However, since it is non-blocking, the target can do something else, e.g. sending
- Also Response exclusion rule must be honored!

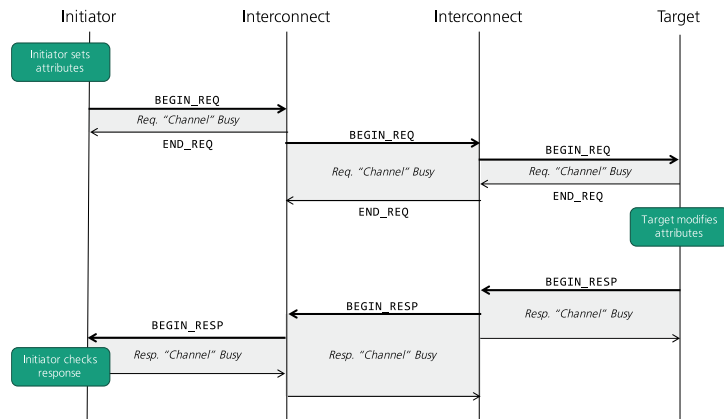


# Base Protocol Rules: Causality with nb\_transport

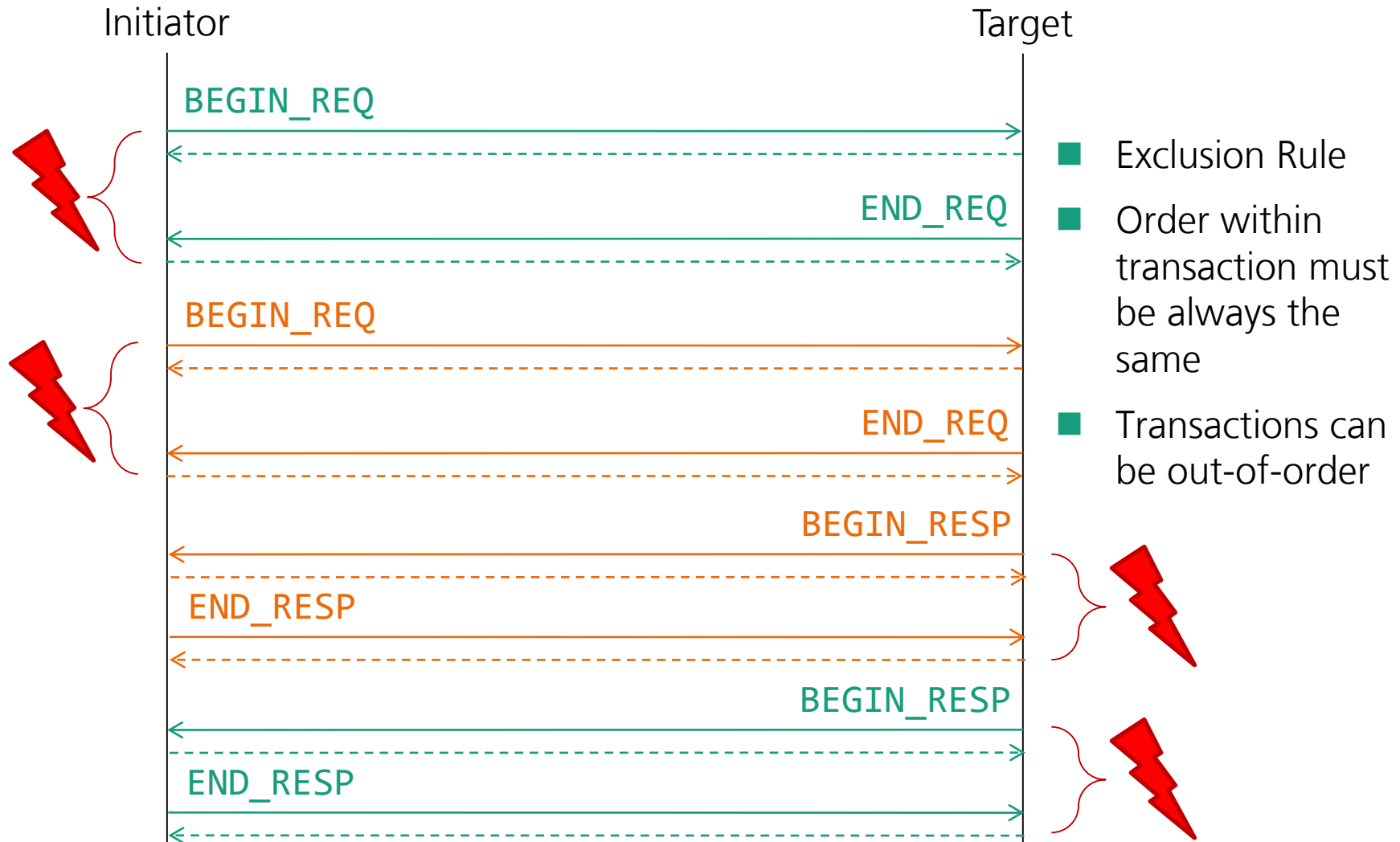


## Base Protocol Rules: Causality with nb\_transport

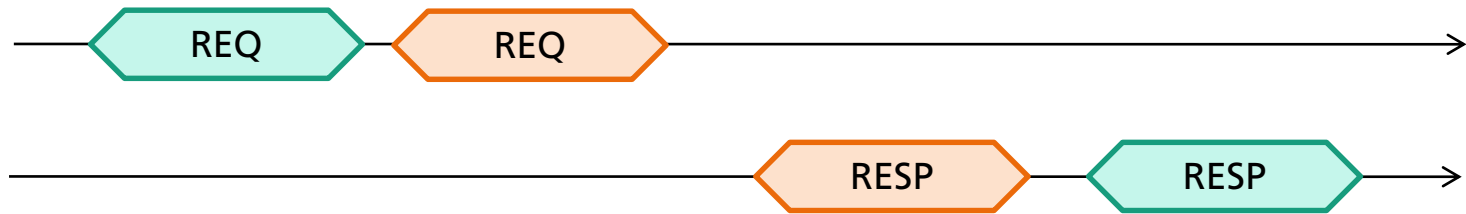
- **BEGIN\_REQ** and **BEGIN\_RESP** are propagated from end-to-end
- **END\_REQ** and **END\_RESP** are not propagated → they are local to each hop
- The **END\_REQ** and **END\_RESP** phases indicate that the requests and response “channels” are no longer busy and may be used for the next request or response respectively
- Initiator should not check the transaction payload until it has received the response

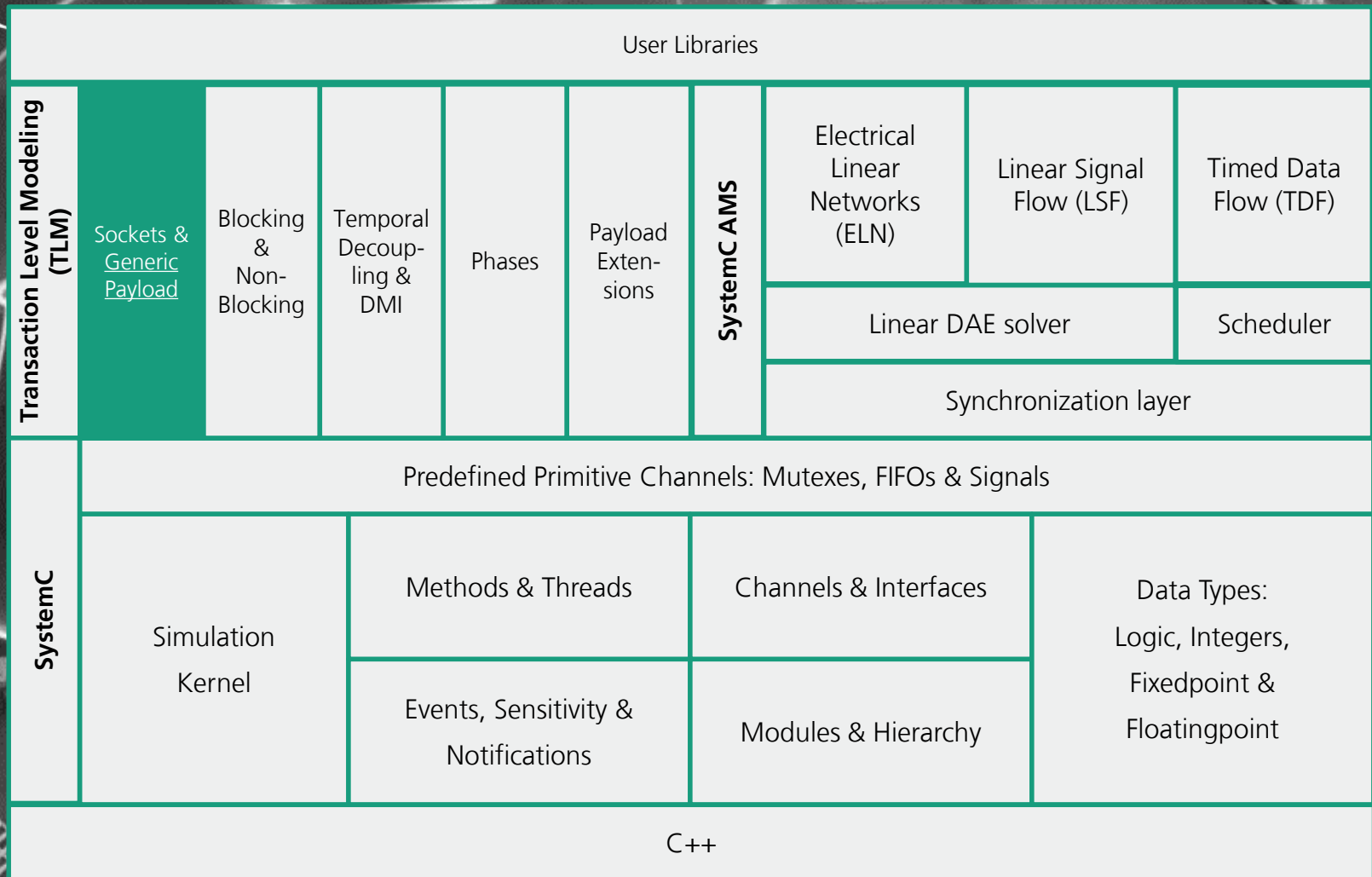


# Base Protocol Rules: Pipelining of Transactions



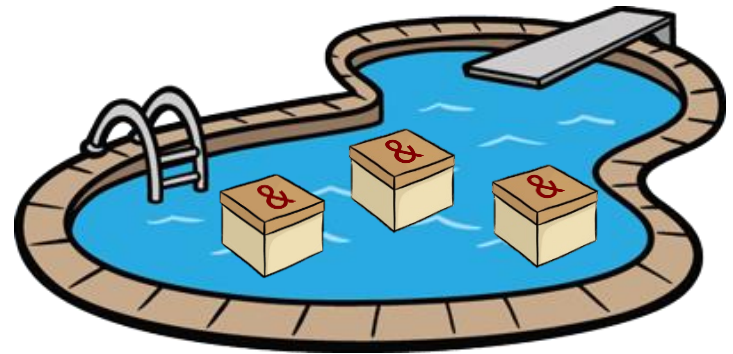
# Alternative View





# Generic Payload Pools (a.k.a Memory Management)

- Allocating of generic payload objects consumes wall-clock time
- Idea: do not allocate for each transaction a new generic payload → **Reuse**
  - A *Memory Manager* handles the generic payload objects in a pool
  - Before sending a transaction a payload object is **allocated** from the pool
  - Modules that send (or receive) this payload object must increase a reference count (**acquire**), which signalizes the memory manager that this object is still in use.
  - If a module is finished with the payload object, the reference count is decreased (**release**)
  - If the reference count is 0 the payload is freed and goes back into the pool.
  - If all transactions in the pool are in use a new one is generated





# Memory Manager in TLM

- The memory manager is not part of the SystemC TLM Standard
- The TLM Standard provides only an interface class:

```
class tlm_mm_interface
{
    public:
    virtual void free(tlm::tlm_generic_payload*) = 0;
    virtual ~tlm_mm_interface(){}
}
```

- The implementation of the memory manager is up to the end user
- Virtual Platform tools like Synopsys Platform Architect have clever implementations

# Example Memory Manager

Use code on github:

[https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/tlm\\_memory\\_manager](https://github.com/TUK-SCVP/SCVP.artifacts/tree/master/tlm_memory_manager)

```
class MemoryManager : public tlm::tlm_mm_interface {
private:
    unsigned int numberOfAllocations;
    unsigned int numberOfFrees;
    std::vector<tlm::tlm_generic_payload*> freePayloads;

public:
    MemoryManager(): numberOfAllocations(0), numberOfFrees(0) {}

    ~MemoryManager(){
        for(tlm::tlm_generic_payload* payload: freePayloads) {
            delete payload;
            numberOfFrees++;
        }
    }

    tlm::tlm_generic_payload* MemoryManager::allocate(){
        if(freePayloads.empty()) {
            numberOfAllocations++;
            return new tlm::tlm_generic_payload(this);
        } else {
            tlm::tlm_generic_payload* result = freePayloads.back();
            freePayloads.pop_back();
            return result;
        }
    }

    void free(tlm::tlm_generic_payload* payload) {
        payload->reset(); //clears all fields and extensions
        freePayloads.push_back(payload);
    }
};
```

Transaction Pool  
implemented as  
`std::vector`

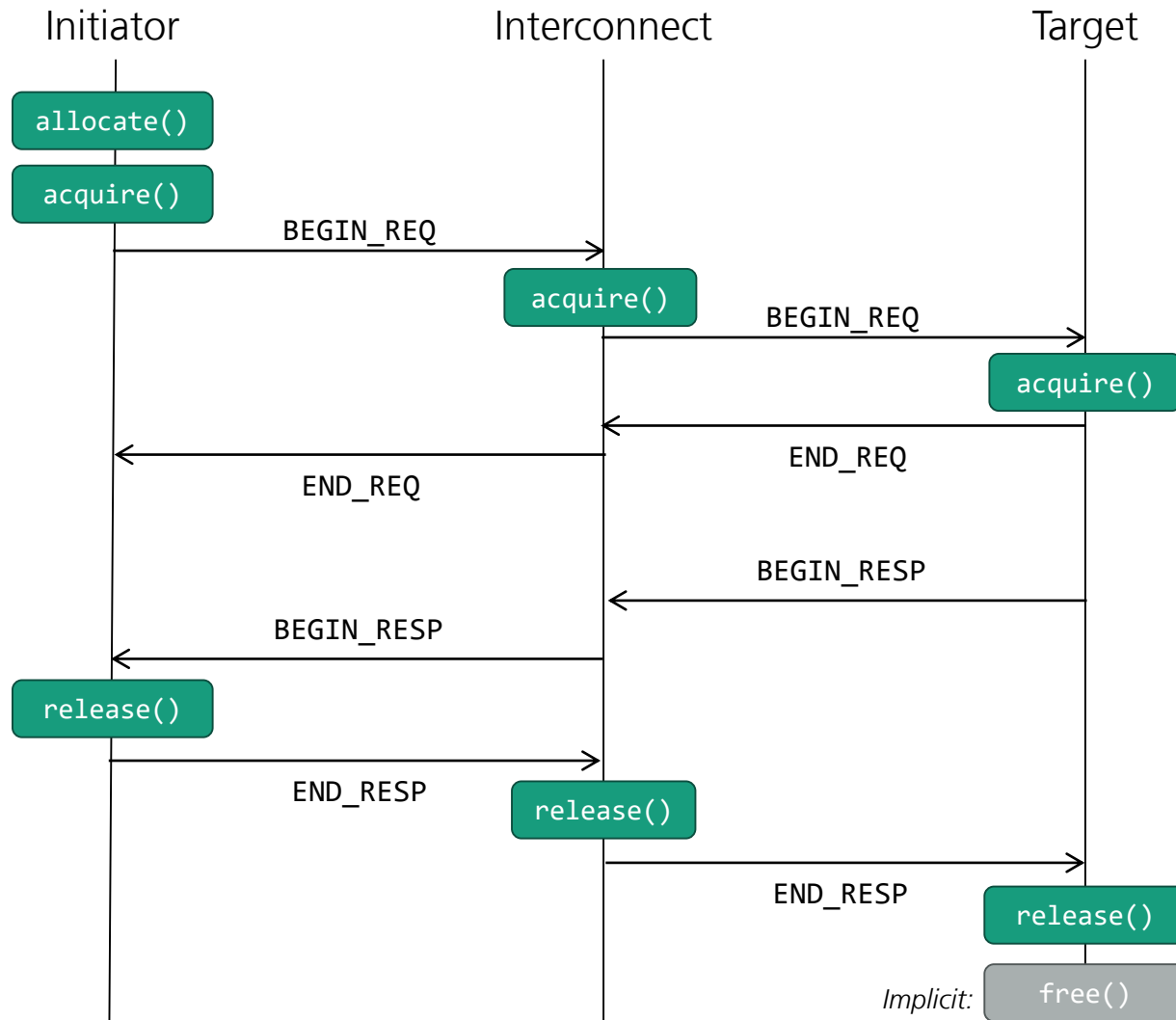
Cleanup  
transaction pool

Allocate new  
payload

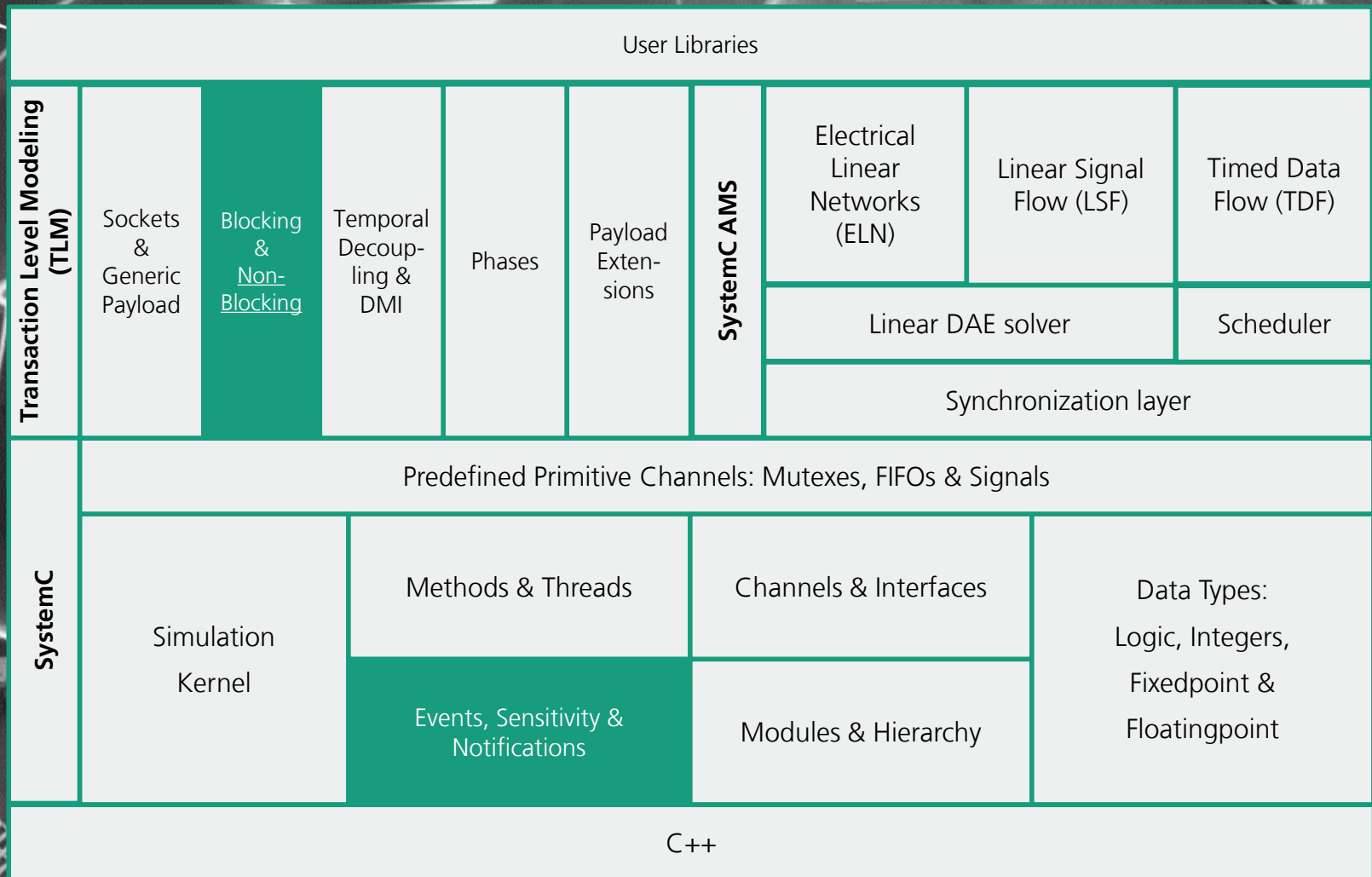
Reuse Payload  
and remove  
from pool

Add transaction  
back to the pool

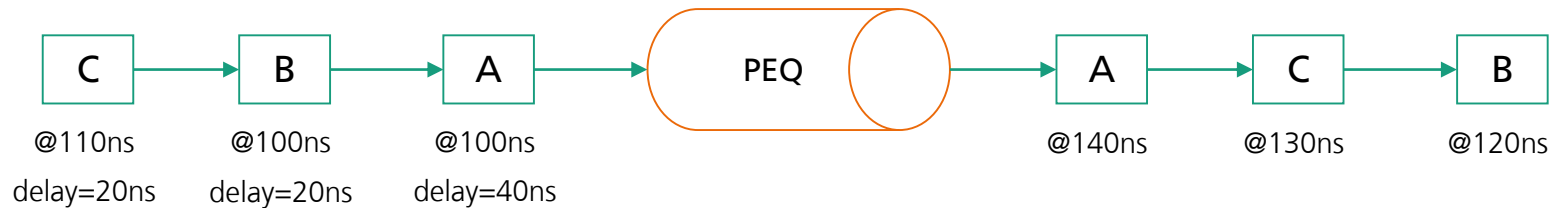
# Usage of Memory Manager



- **b\_transport**  
a.k.a. LT may use a memory manager
- **nb\_transport**  
a.k.a. AT must use a memory manager

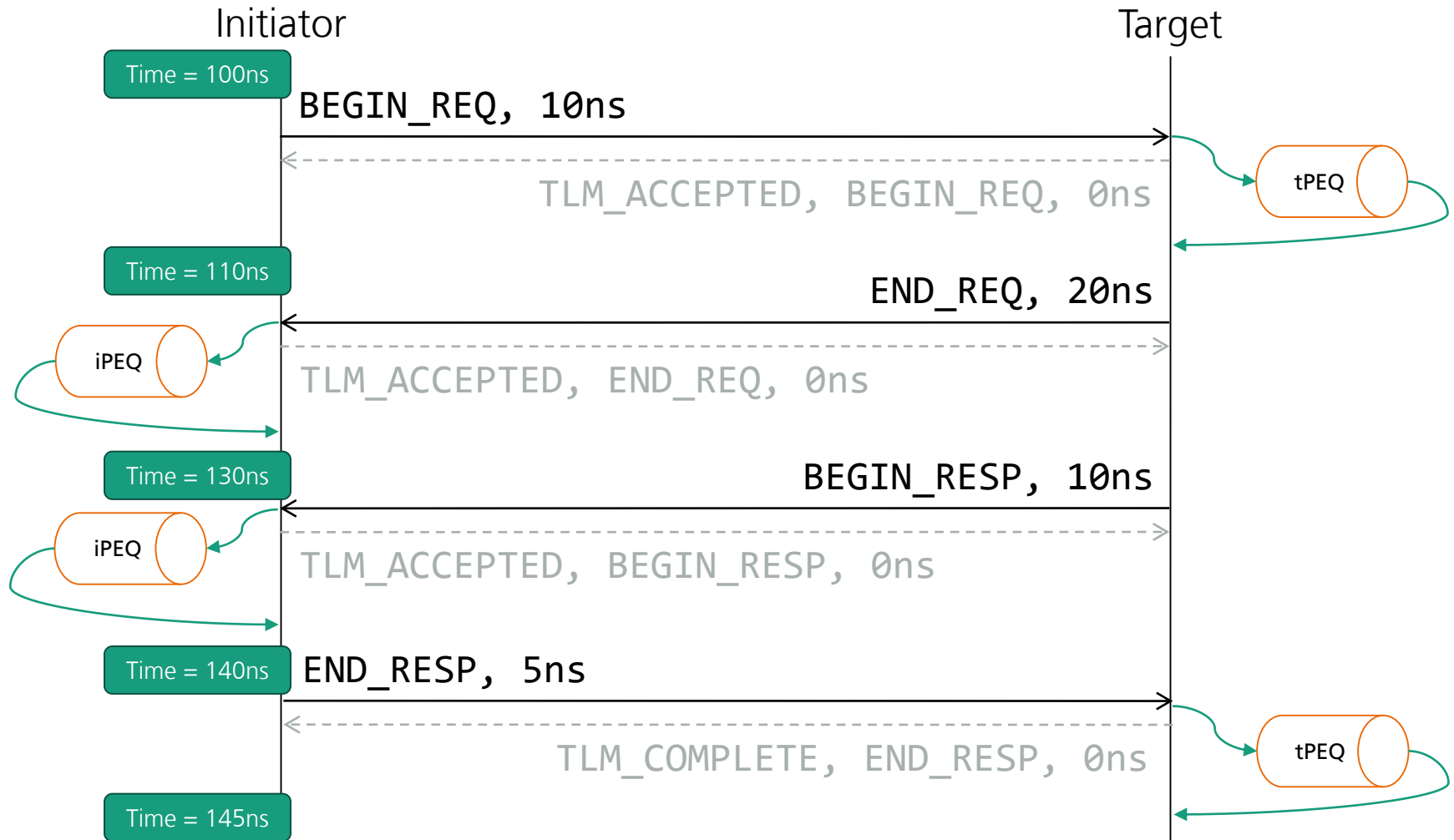


# The Payload Event Queue

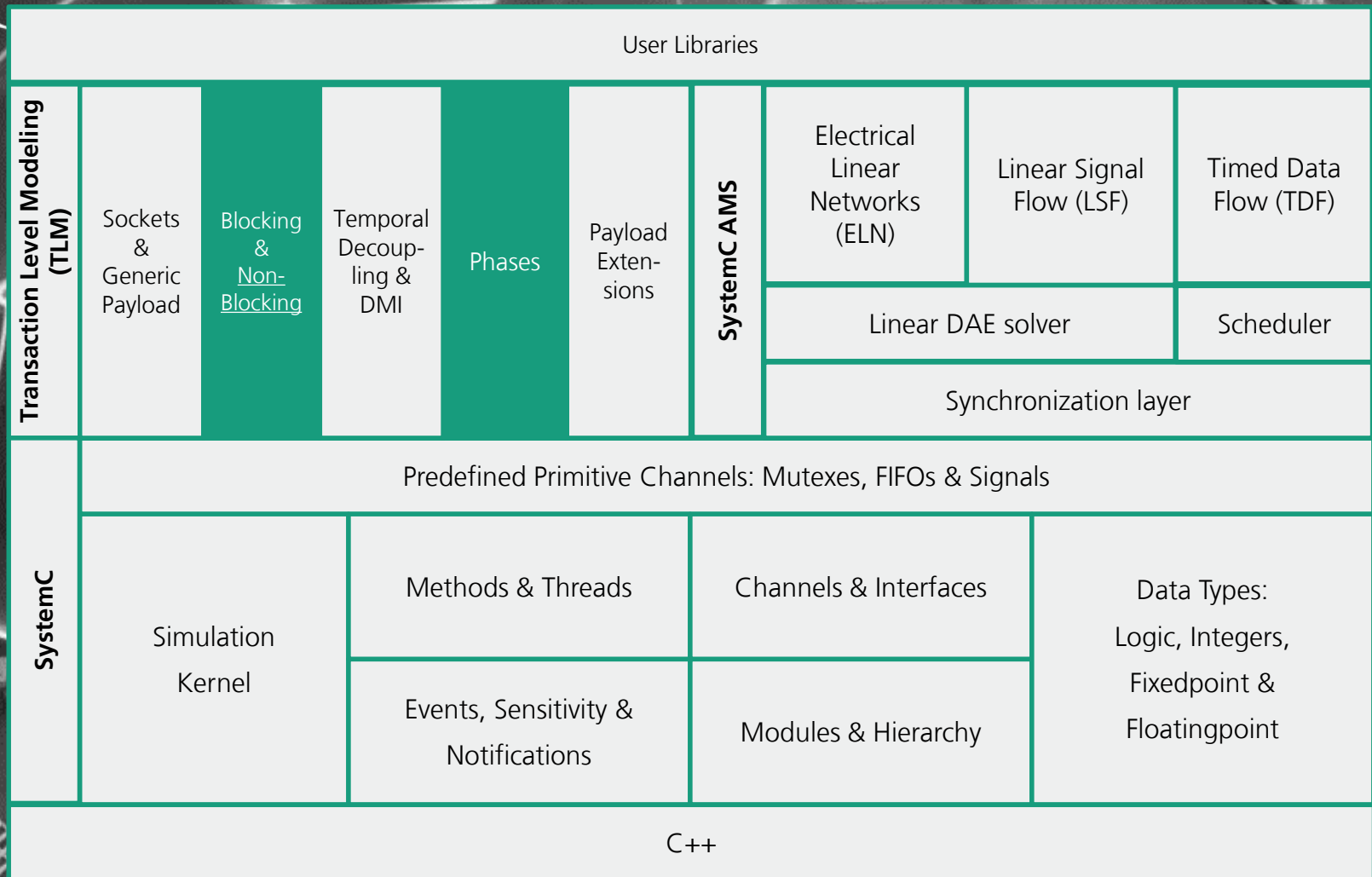


- AT models usually synchronize incoming calls with the simulation time
- Timing annotation is similar to `b_transport`:
  - *Hey target, please pretend that you received this message 10ns in the future*
- An AT component is usually not calling `wait()` statements for synchronizing with time, it rather posts the incoming transaction into a *Payload Event Queue* (PEQ). The PEQ internally synchronizes with the simulation time and calls a callback function in order to process the transaction when the time is reached.
- A component receives a transaction that should be processed in the future, so it puts the transaction into a PEQ in order to process it when time is reached

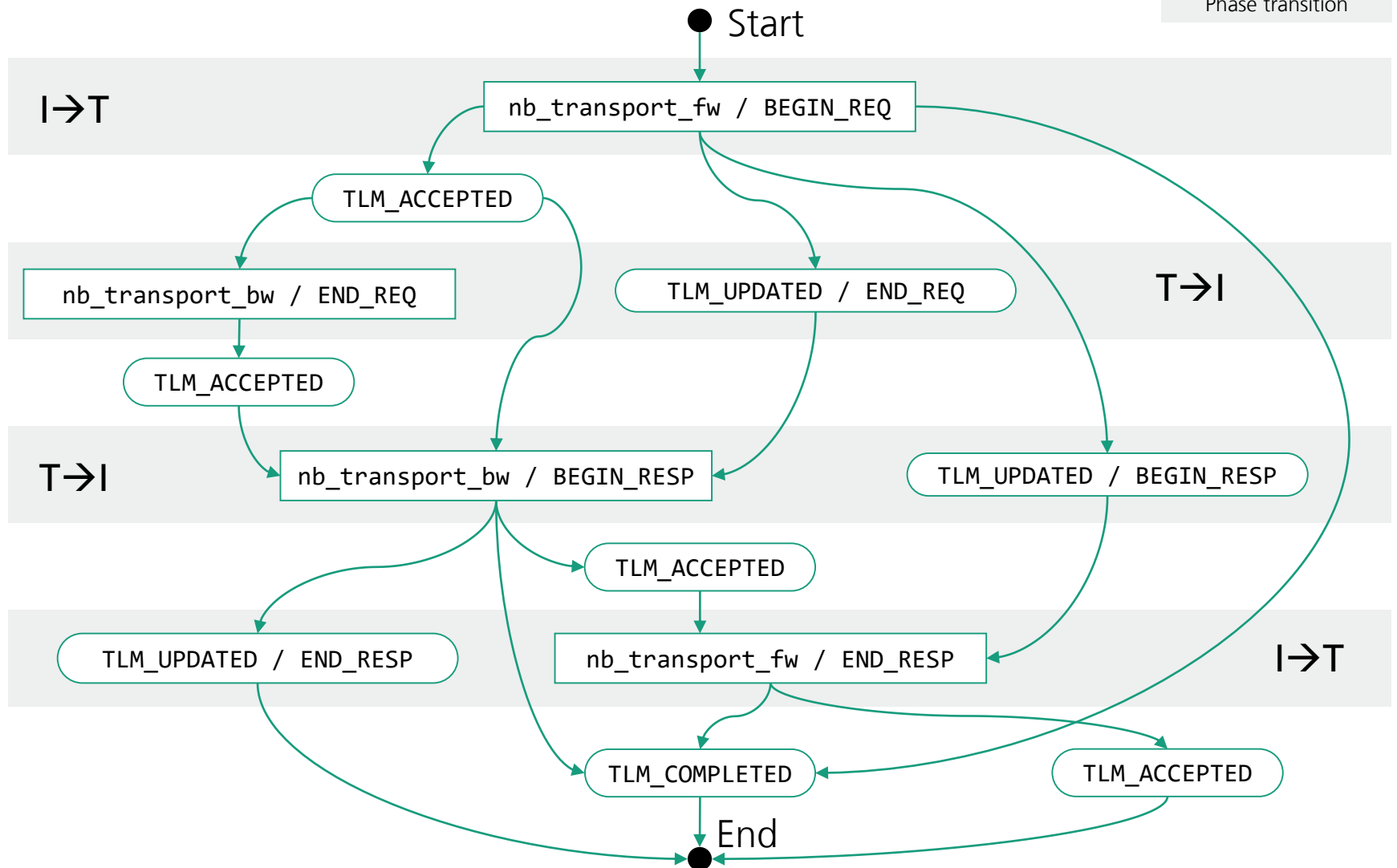
# Timing Annotation with AT Models







# Full Overview of AT Protocol Possibilities

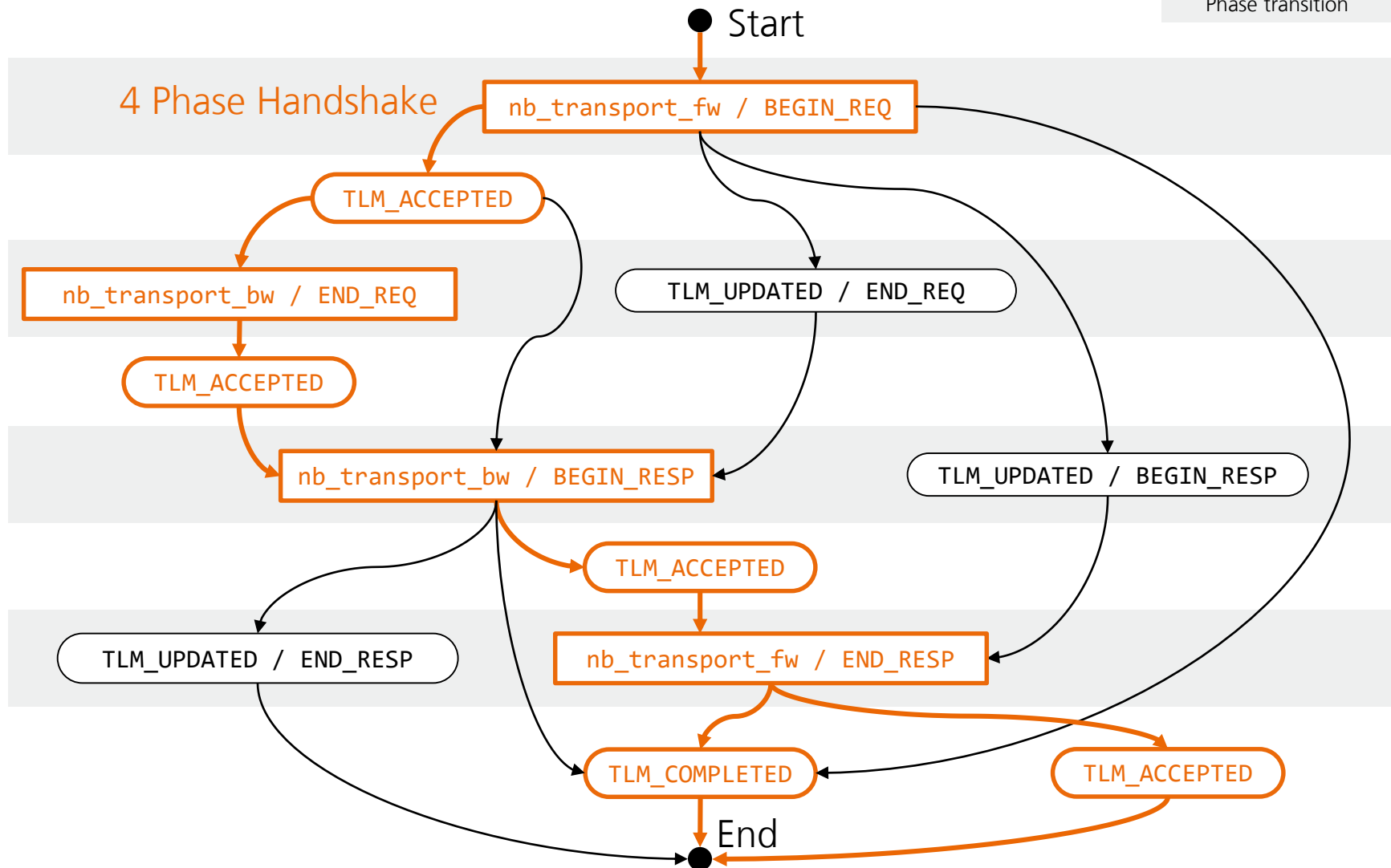


# Full Overview of AT Protocol Possibilities

call

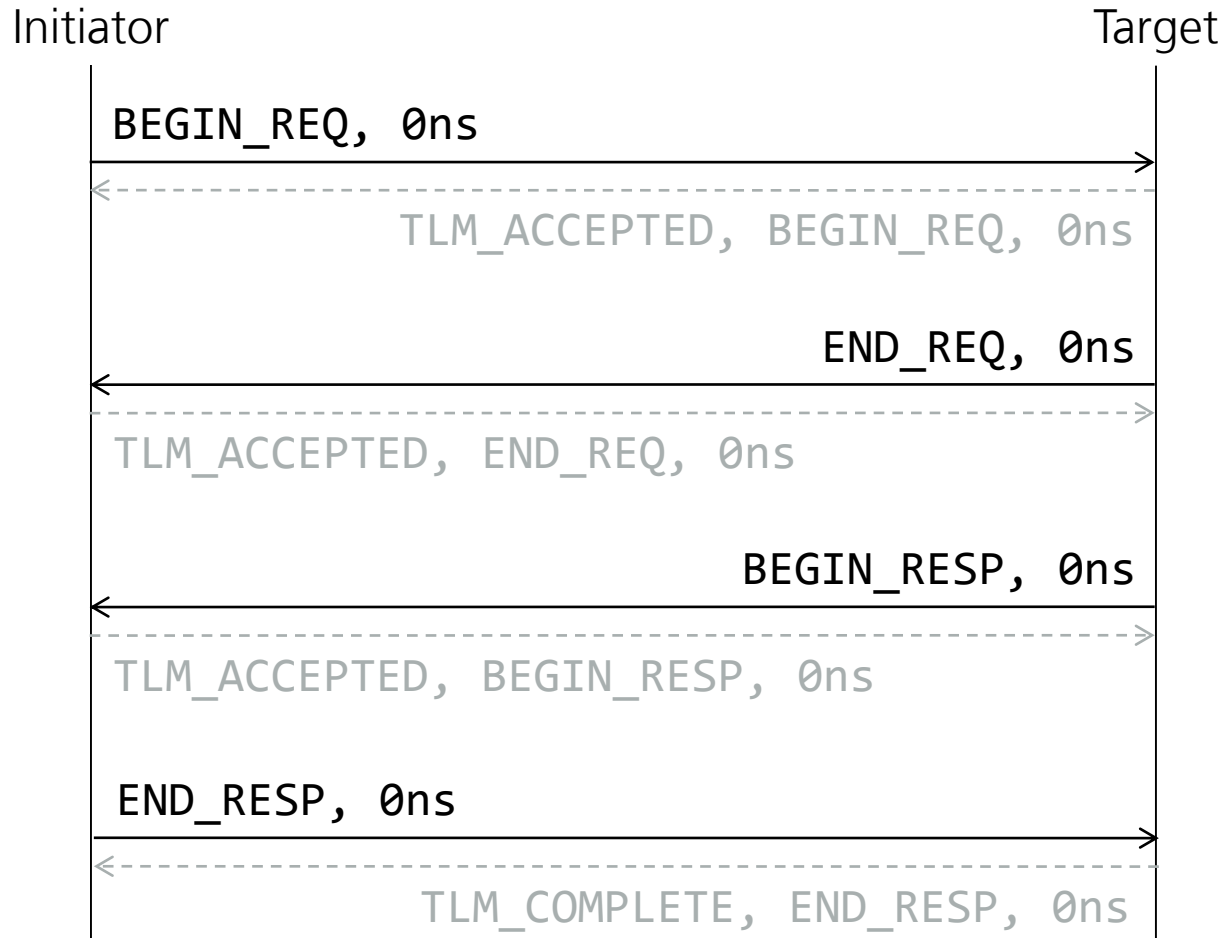
return

Phase transition



29

# Base Protocol Rules [1]: 4 Phase Handshake



# 4 Phase Handshake: Initiator

```
class Initiator: public sc_module, public tlm::tlm_bw_transport_if<> {
public:
    tlm::tlm_initiator_socket<> socket;
    MemoryManager mm;
    int data[16];
    tlm::tlm_generic_payload* requestInProgress;
    sc_event endRequest;
    tlm_utils::peq_with_cb_and_phase<Initiator> peq;

    SC_CTOR(Initiator): socket("socket"),
        requestInProgress(0),
        peq(this, &Initiator::peqCallback)

    {
        socket.bind(*this);
        SC_THREAD(process);
    }

protected:
    void process() {
        tlm::tlm_generic_payload* trans;
        tlm::tlm_phase phase;
        sc_time delay;

        for (int i = 0; i < 100; i++) {
            trans = mm.allocate();
            trans->acquire();

            trans->set_command(...);
            ...
            trans->set_response_status(tlm::TLM_INCOMPLETE_RESPONSE);

            if (requestInProgress) {
                wait(endRequest);
            }
            requestInProgress = trans;
            phase = tlm::BEGIN_REQ;
            delay = ...;

            tlm::tlm_sync_enum status;
            status = socket->nb_transport_fw( *trans, phase, delay );
            ...
            wait(randomDelay());
        }
    }
};
```

Generate a sequence of random transactions

Grab a new transaction from the memory manager pool

BEGIN\_REQ/END\_REQ exclusion rule

```
virtual tlm::tlm_sync_enum nb_transport_bw(tlm::tlm_generic_payload& trans,
                                           tlm::tlm_phase& phase,
                                           sc_time& delay)
{
    peq.notify(trans, phase, delay);
    return tlm::TLM_ACCEPTED;
}

void peqCallback(tlm::tlm_generic_payload& trans,
                 const tlm::tlm_phase& phase)
{
    if (phase == tlm::END_REQ || ...)
    {
        requestInProgress = 0;
        endRequest.notify();
    }
    else if (phase == tlm::BEGIN_RESP)
    {
        checkTransaction(trans);

        tlm::tlm_phase fw_phase = tlm::END_RESP;
        sc_time delay = sc_time(...);

        socket->nb_transport_fw( trans, fw_phase, delay );

        trans.release();
    }
    else if (phase == tlm::BEGIN_REQ || phase == tlm::END_RESP)
    {
        SC_REPORT_FATAL(name(), "Illegal transaction phase received");
    }
}
};
```

Queue the transaction into the peq until the annotated time has elapsed

Payload event queue callback

Wake-up suspended main process

Do something with transaction

Allow MM to free the transaction object

# 4 Phase Handshake: Target

```

class Target: public sc_module, public tlm::tlm_fw_transport_if<> {
public:
    tlm::tlm_target_socket<> socket;
    tlm::tlm_generic_payload* transactionInProgress;
    sc_event targetDone;
    bool responseInProgress;
    tlm::tlm_generic_payload* nextResponsePending;
    tlm::tlm_generic_payload* endRequestPending;
    tlm_utils::peq_with_cb_and_phase<Target> peq;

    SC_CTOR(Target) : socket("socket"),
        transactionInProgress(0),
        responseInProgress(false),
        nextResponsePending(0),
        endRequestPending(0),
        peq(this, &Target::peqCallback)
    {
        socket.bind(*this);

        SC_METHOD(executeTransactionProcess);
        sensitive << targetDone; dont_initialize();
    }
    ...
    tlm::tlm_sync_enum nb_transport_fw(tlm::tlm_generic_payload& trans,
        tlm::tlm_phase& phase,
        sc_time& delay)
    {
        peq.notify( trans, phase, delay);
        return tlm::TLM_ACCEPTED;
    }

    void peqCallback(tlm::tlm_generic_payload& trans,
        const tlm::tlm_phase& phase)
    {
        sc_time delay;

        if(phase == tlm::BEGIN_REQ) {
            trans.acquire();

            if (!transactionInProgress) {
                sendEndRequest(trans);
            }
            else {
                endRequestPending = &trans;
            }
        }
    }
}

```

Increment the transaction reference count

Put back-pressure on initiator by deferring END\_REQ

```

else if (phase == tlm::END_RESP) {
    ...
    transactionInProgress = 0;
    responseInProgress = false;

    if (nextResponsePending) {
        sendResponse(*nextResponsePending);
        nextResponsePending = 0;
    }

    if (endRequestPending) {
        sendEndRequest(*endRequestPending);
        endRequestPending = 0;
    }
}
else // tlm::END_REQ or tlm::BEGIN_RESP
{
    SC_REPORT_FATAL(name(), "Illegal transaction phase received");
}

void sendEndRequest(tlm::tlm_generic_payload& trans)
{
    tlm::tlm_phase bw_phase;
    sc_time delay;

    bw_phase = tlm::END_REQ;
    delay = ...; // Accept delay

    tlm::tlm_sync_enum status;
    status = socket->nb_transport_bw( trans, bw_phase, delay );

    delay = delay + randomDelay();
    targetDone.notify( delay );

    assert(transactionInProgress == 0);
    transactionInProgress = &trans;
}
...

```

Flag must only be cleared when END\_RESP is sent

Target itself is now clear to issue the next BEGIN\_RESP

unlock the initiator by issuing END\_REQ

Queue internal event to mark beginning of response



# 4 Phase Handshake: Target

```
void executeTransactionProcess()
{
    executeTransaction(*transactionInProgress);

    if (responseInProgress)
    {
        ...
        nextResponsePending = transactionInProgress;
    }
    else
    {
        sendResponse(*transactionInProgress);
    }
}
```

Method process that runs on  
targetDone event

Target must honor  
BEGIN\_RESP/END\_RE  
SP exclusion rule

```
void sendResponse(tlm::tlm_generic_payload& trans)
{
    tlm::tlm_sync_enum status;
    tlm::tlm_phase bw_phase;
    sc_time delay;

    responseInProgress = true;
    bw_phase = tlm::BEGIN_RESP;
    delay = SC_ZERO_TIME;
    status = socket->nb_transport_bw( trans, bw_phase, delay );

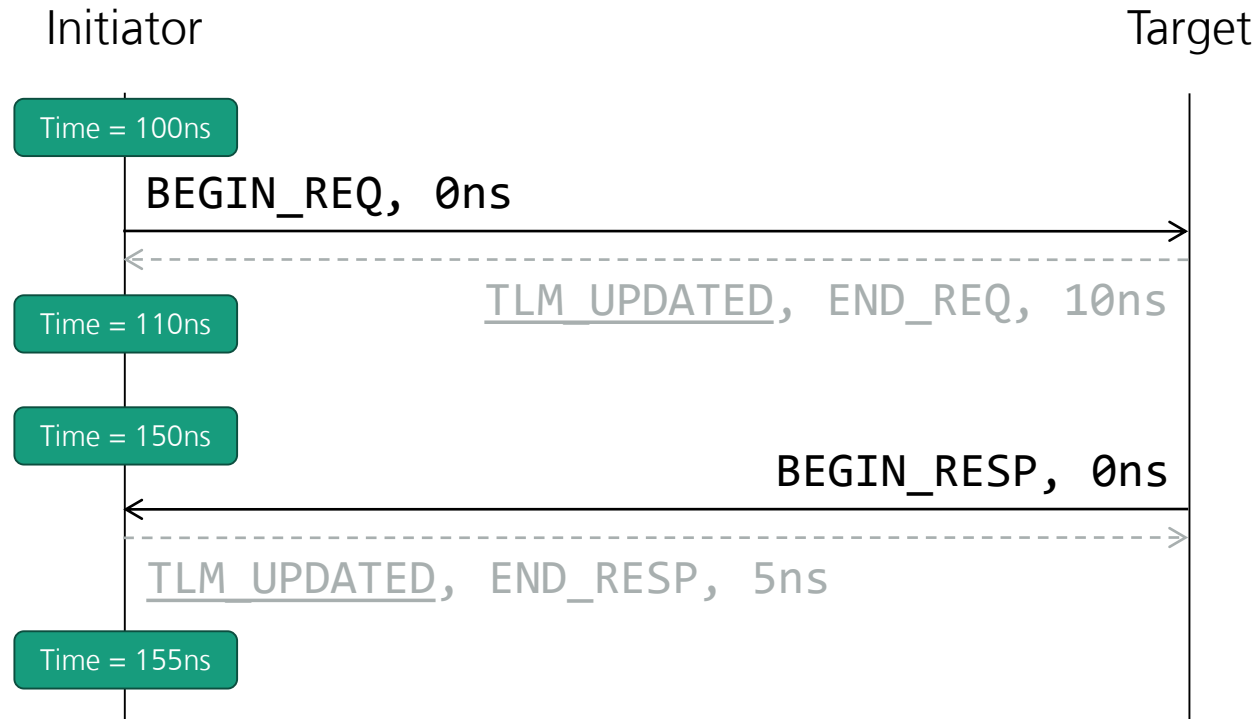
    ...

    trans.release();
}
...
};
```

Function for sending  
the response

Tell the memory  
manager to free  
transaction

# Base Protocol Rules [2]: Using the Return Paths



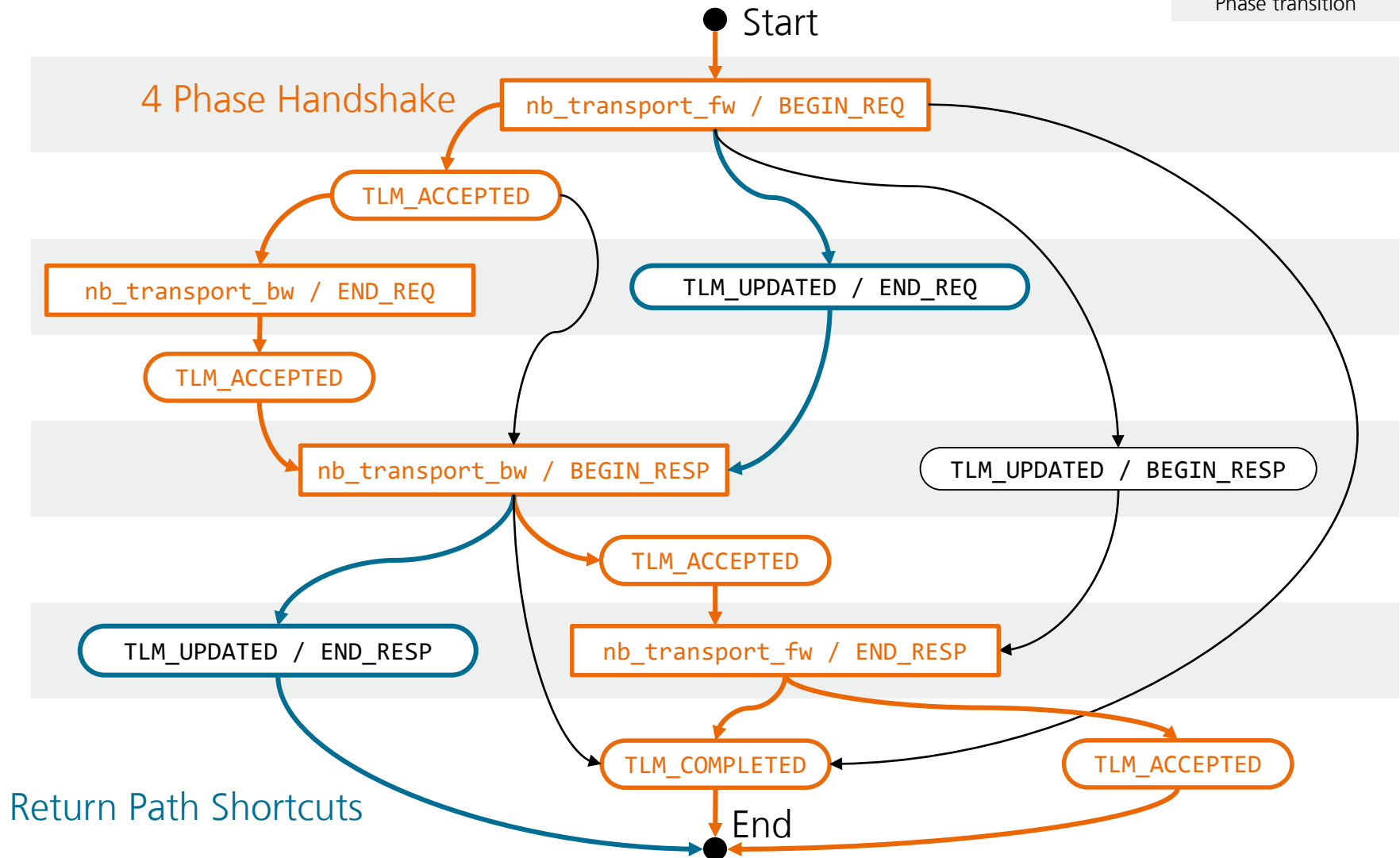
- The FW return path can be used as an alternative to the BW path
- The BW return path can be used as an alternative to the FW path
  - State must be set to **TLM\_UPDATED**
- Timing must be increased

# Full Overview of AT Protocol Possibilities

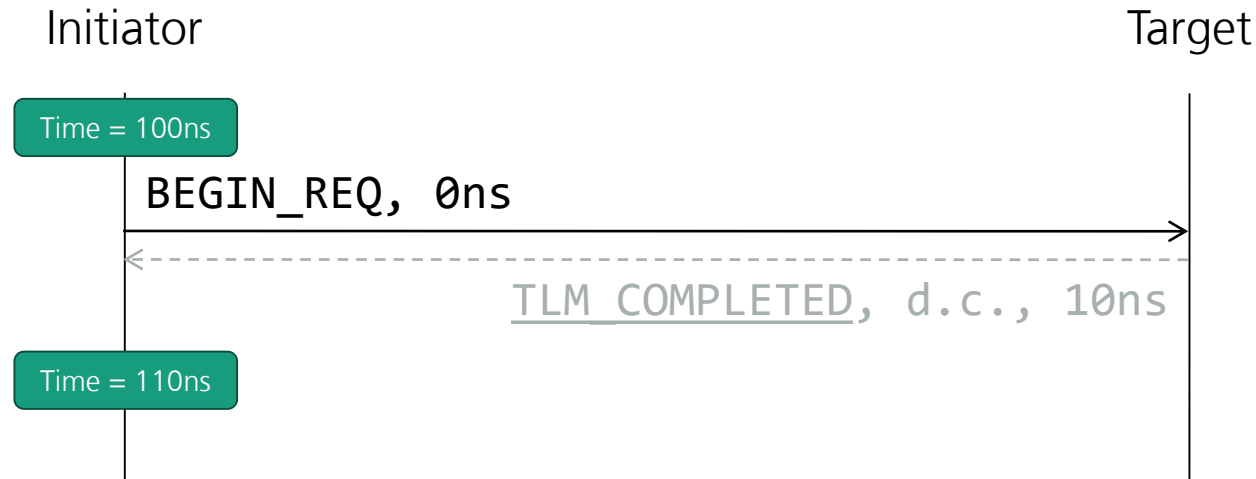
call

return

Phase transition

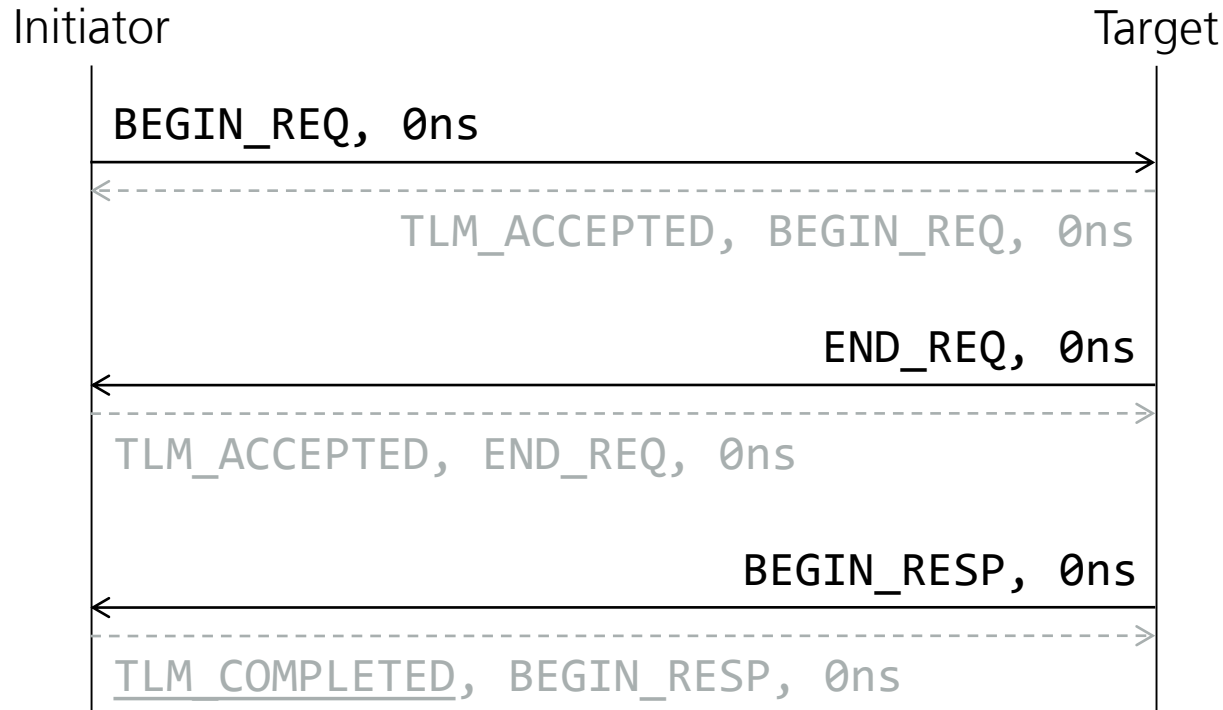


# Base Protocol Rules [4]: Early Completion



- The initiator sends **BEGIN\_REQ** and the target immediately returns **TLM\_COMPLETED** (useful for modelling simple I/O, where no ACK is required)
- The targets pre-empt any further communication by signaling a so called *Early completion*
- Initiator should immediacy check the response status of the generic payload object
- With **TLM\_COMPLETED** the caller should always ignore the phase argument

# Base Protocol Rules [4]: Early Completion



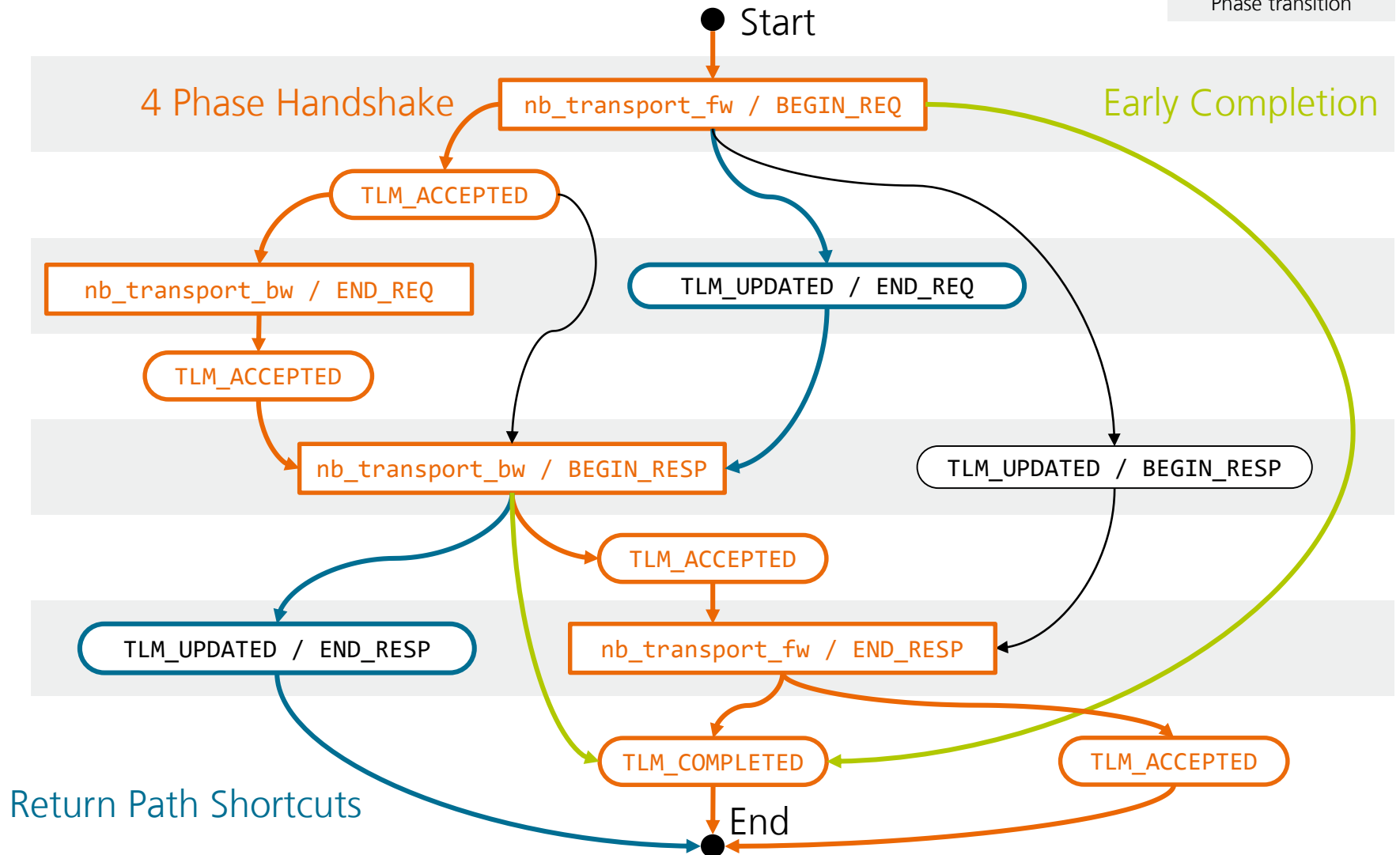
- The initiator can early complete the transaction by returning **TLM\_COMPLETED**
- The initiator pre-empts therefore any further communication

# Full Overview of AT Protocol Possibilities

call

return

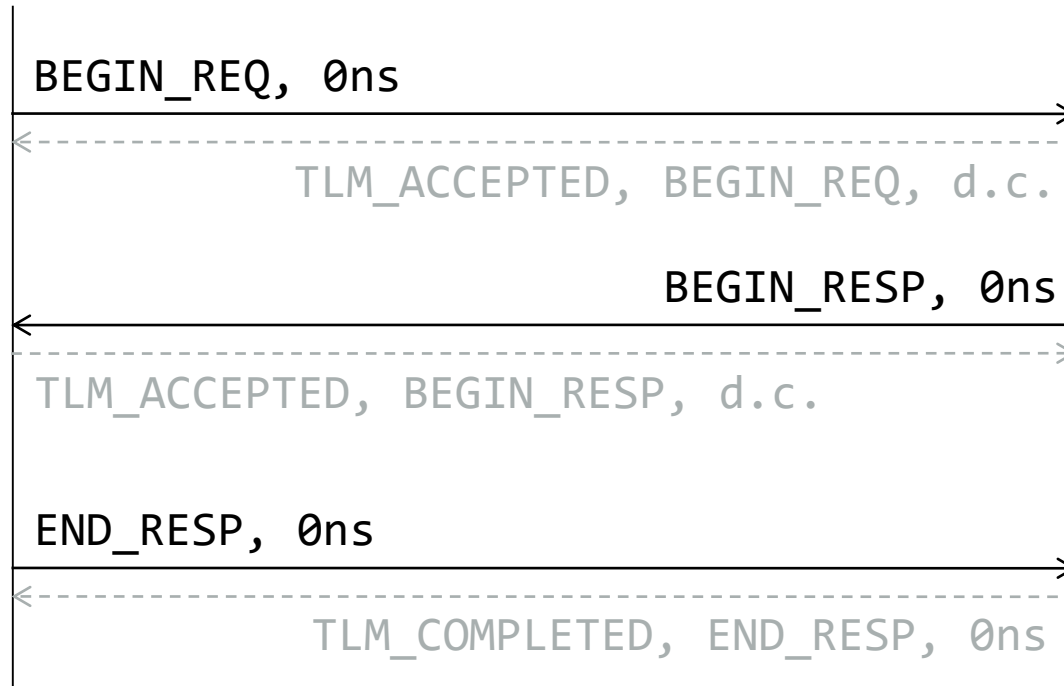
Phase transition



# Base Protocol Rules [3]: Skip END\_REQ

Initiator

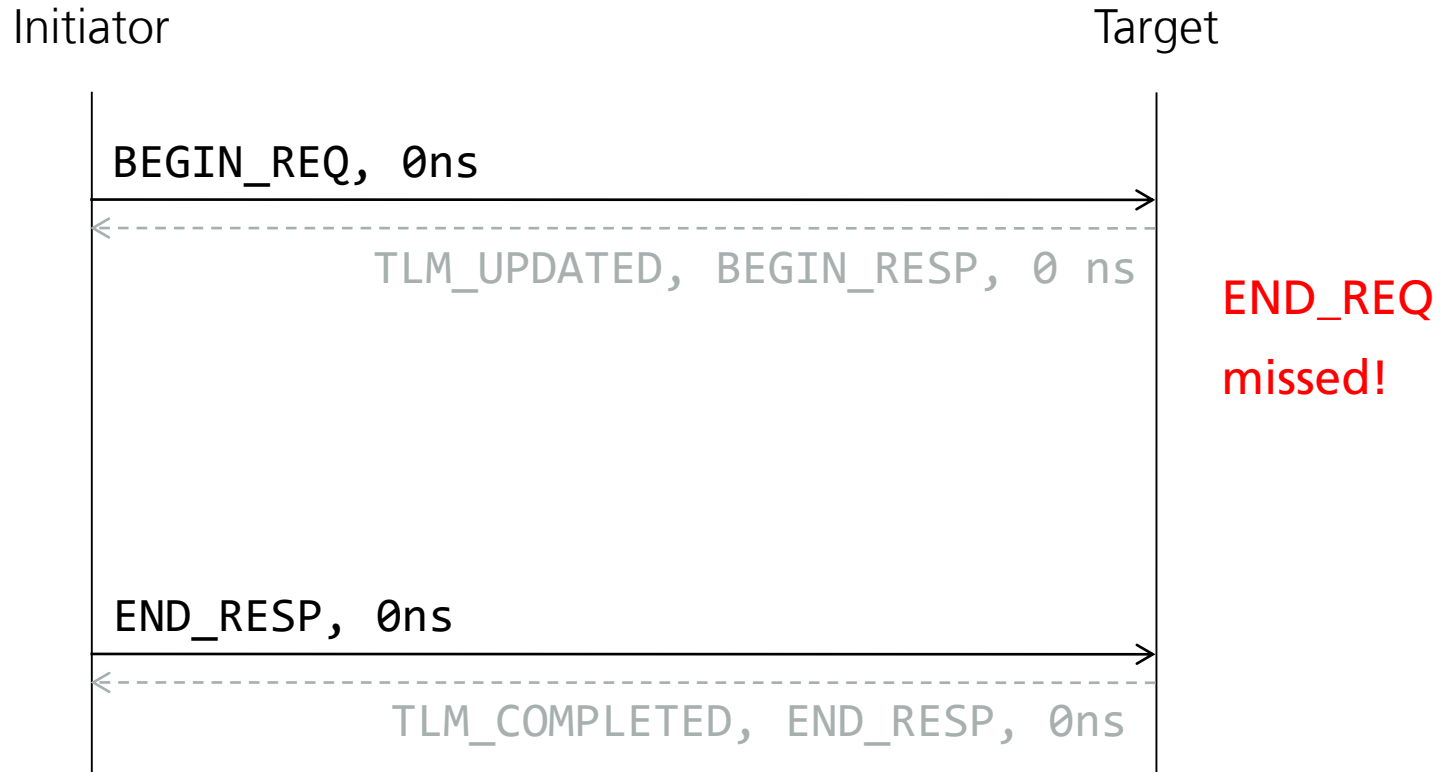
Target



**END\_REQ  
missed!**

- END\_REQ is pre-empted by the target sending BEGIN\_RESP in the next BW call

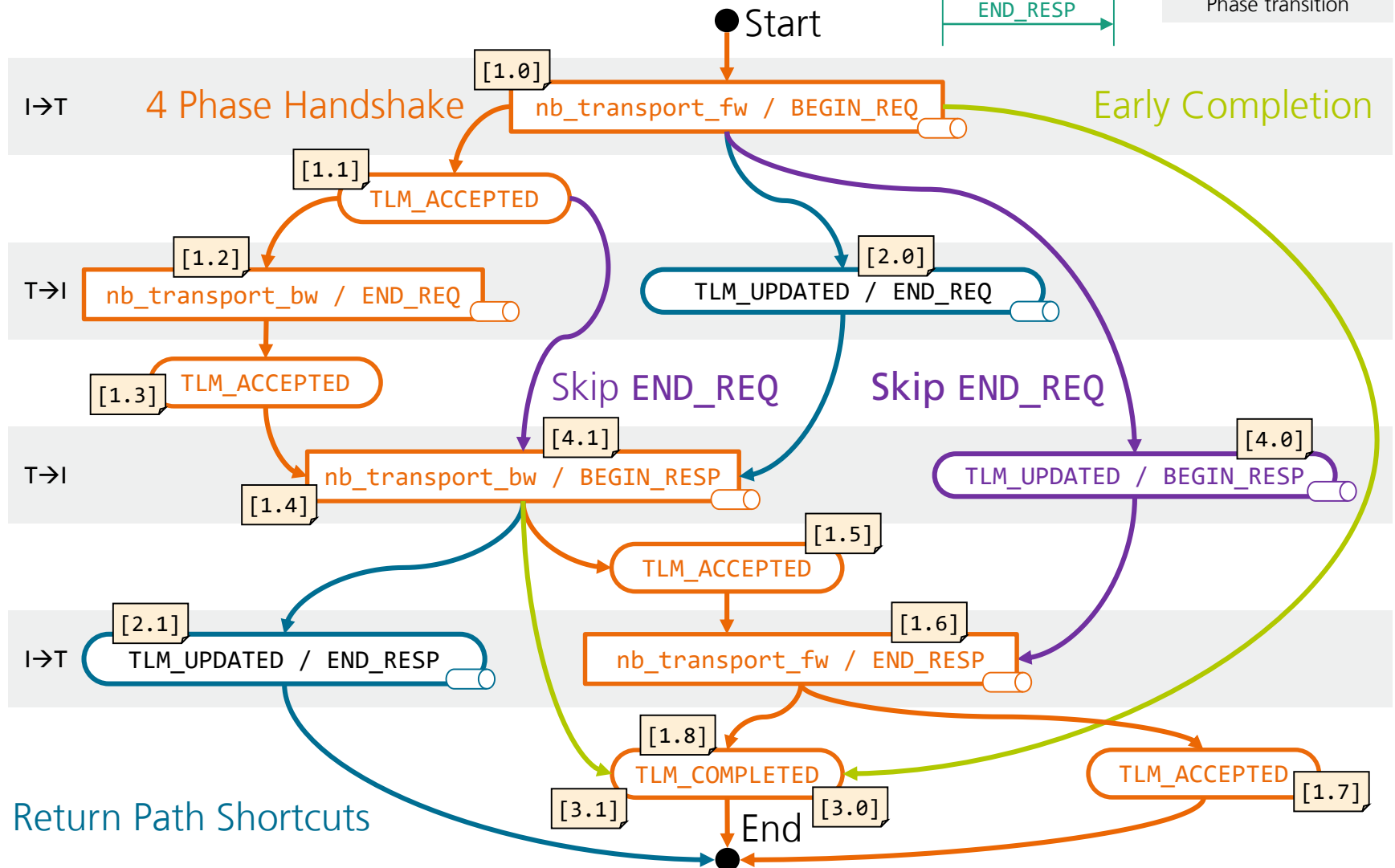
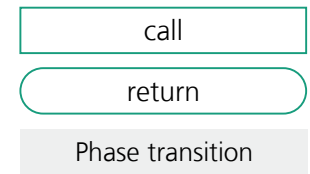
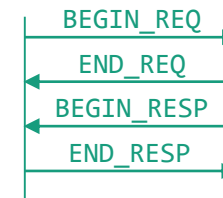
# Base Protocol Rules (7): Skip END\_REQ (Shortcut)

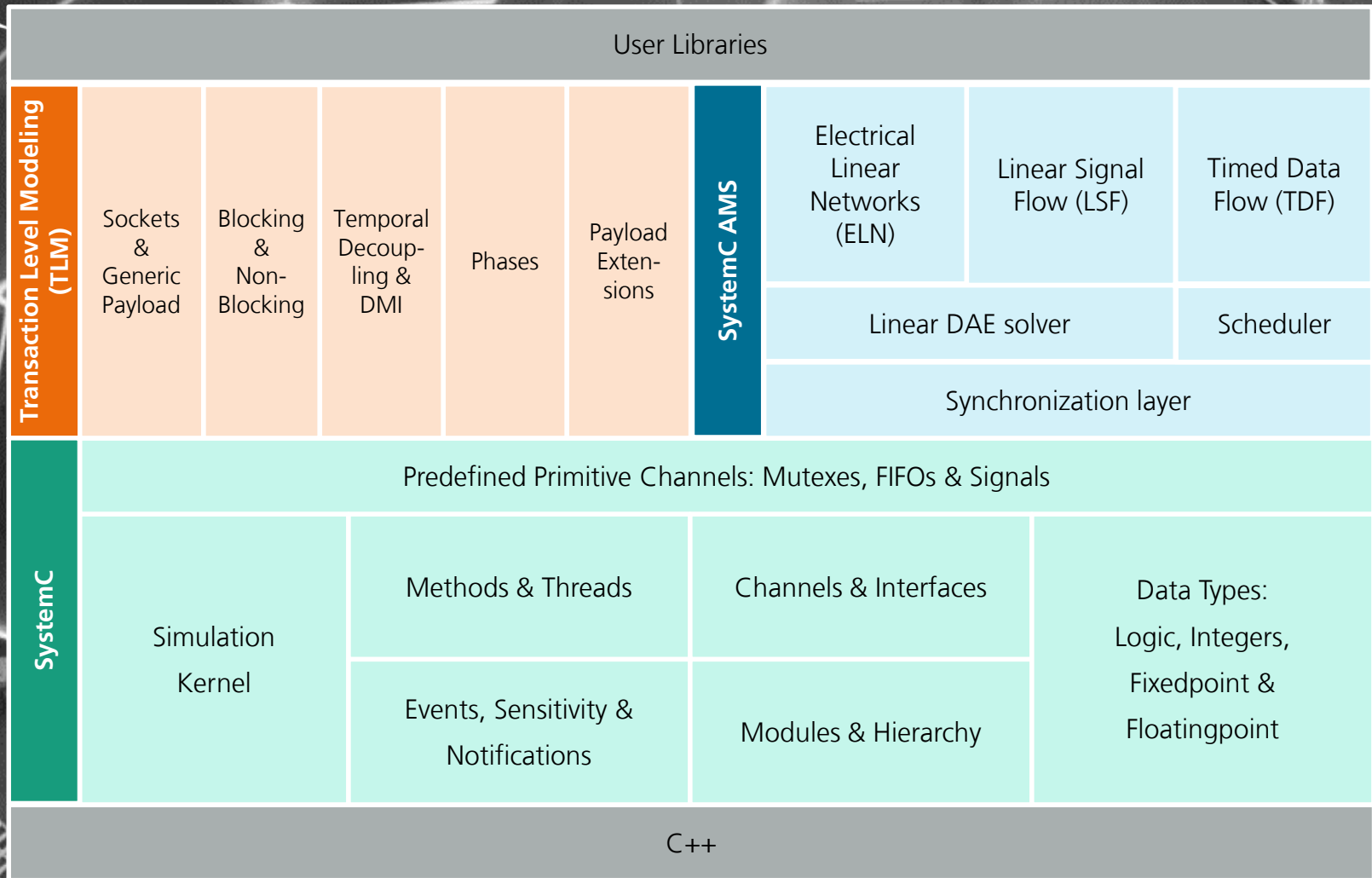


- END\_REQ is pre-empted by the target sending directly BEGIN\_RESP via TLM\_UPDATED over the return path



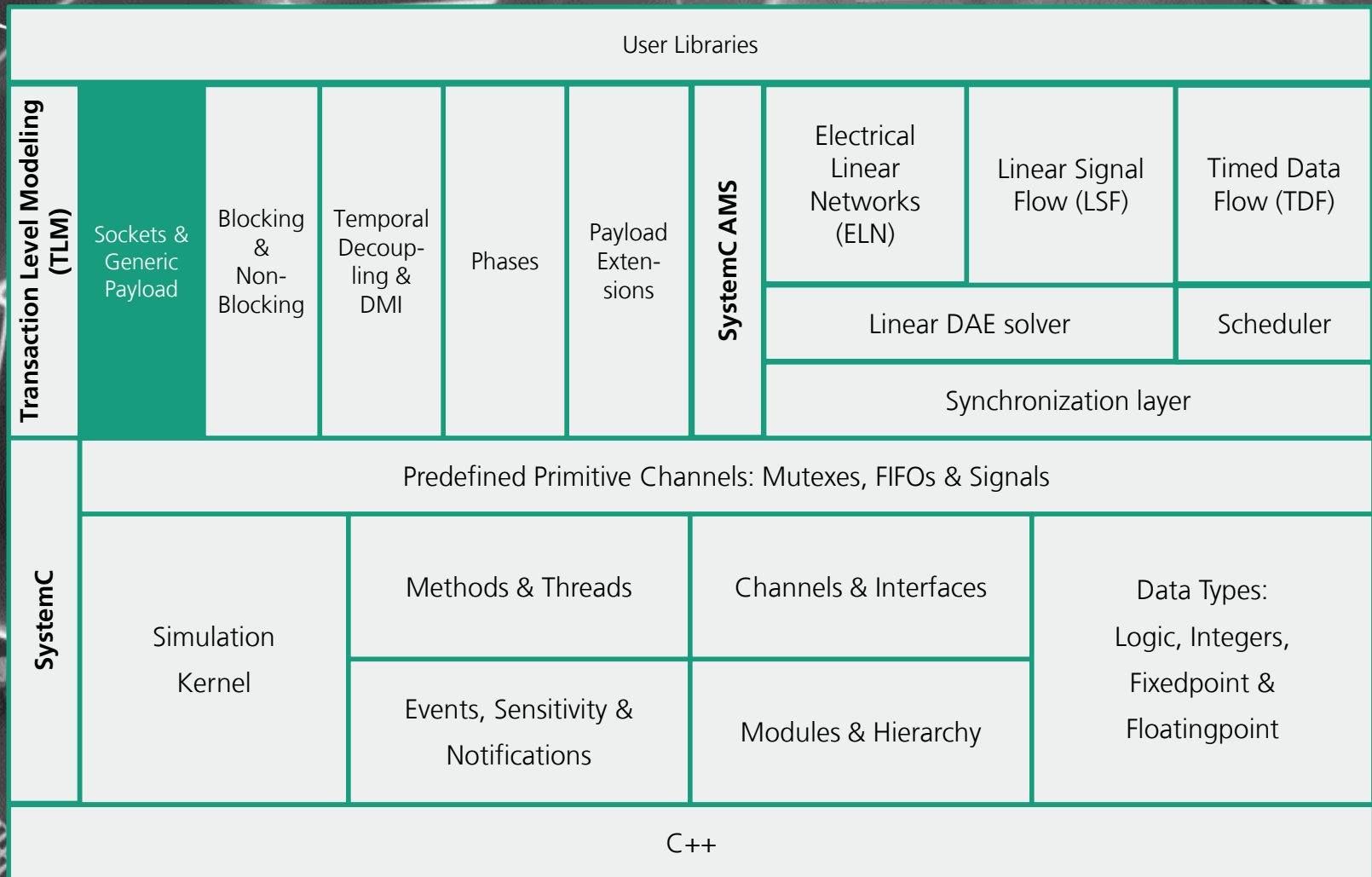
# AT Protocol Cheat Sheet







**TLM Advanced**





# Simple Sockets

- “Simple” because they are easy to use – less to code ...
- Do not bind sockets to objects, instead register methods with each socket
- Dummy method implementations are provided:
  - `get_direct_mem_ptr`
  - `transport_dbg`
  - `invalidate_direct_mem_ptr`
  - `nb_transport_bw`
- Target need only register either `b_transport` or `nb_transport_fw`
- Automatic conversion between blocking and non-blocking

# Simple Sockets: Initiator Example

```
#include "tlm_utils/simple_initiator_socket.h"
...

SC_MODULE(Initiator) {
    public:
        tlm_utils::simple_initiator_socket<Initiator> iSocket;
        ...
    public:
        SC_CTOR(Initiator): iSocket("iSocket"), requestInProgress(0), peq(this, &Initiator::peqCallback)
        {
            iSocket.register_nb_transport_bw(this, &Initiator::nb_transport_bw);

            SC_THREAD(process);
            ...
        }

        void process() {
            ...
        }

        tlm::tlm_sync_enum nb_transport_bw(tlm::tlm_generic_payload& ..., tlm::tlm_phase& ..., sc_time& ...) {
            ...
        }
        ...
};
```

Instantiate simple socket

Register function

Implementation of Function

Try code on github:  
[https://github.com/TUK-SCVP/SCVP.artifacts/blob/master/tlm\\_simple\\_sockets/](https://github.com/TUK-SCVP/SCVP.artifacts/blob/master/tlm_simple_sockets/)

# Simple Sockets: Target Example

```
#include "tlm_utils/simple_target_socket.h"
...
SC_MODULE(Target)
{
    tlm_utils::simple_target_socket<Target> tSocket;
    ...
    SC_CTOR(Target) : tSocket("tSocket"), ...
    {
        tSocket.register_b_transport(this, &Target::b_transport);
        tSocket.register_nb_transport_fw(this, &Target::nb_transport_fw);
    }

    void b_transport(tlm::tlm_generic_payload& trans, sc_time& delay) {
        ...
    }

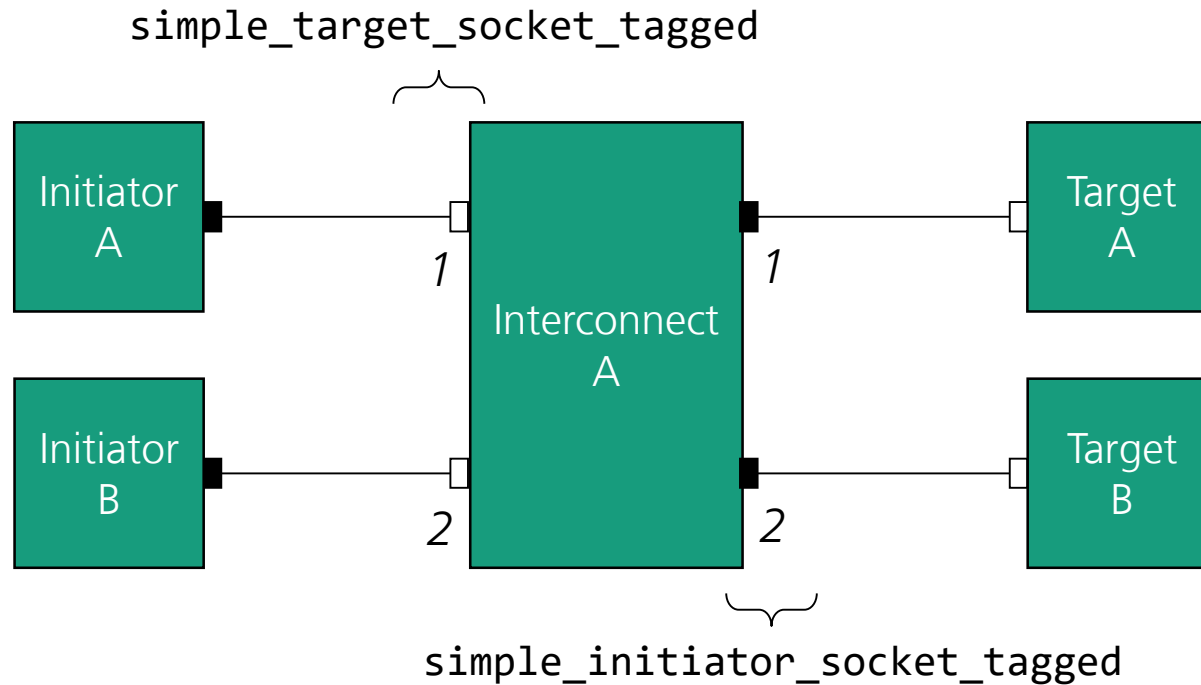
    tlm_sync_enum nb_transport_fw(tlm::tlm_generic_payload& ..., tlm::tlm_phase& ..., sc_time& ...) {
        ...
    }
    ...
};
```

Instantiate simple target socket

Register functions

Implementation of Functions

# Tagged Simple Sockets: Interconnect Example



- Register the same callback method with several sockets
- Distinguish origin of incoming transactions using socket id
- Interconnect is usually template class for number of target and initiator sockets



# Tagged Simple Sockets: Interconnect Example

```
template<unsigned int I, unsigned int T>
SC_MODULE(BUS)
{
    public:
        tlm_utils::simple_target_socket_tagged<BUS> tSocket[T];
        tlm_utils::simple_initiator_socket_tagged<BUS> iSocket[I];

        SC_CTOR(BUS)
        {
            for(unsigned int i = 0; i < T; i++)
            {
                tSocket[i].register_b_transport(this,
                    &BUS::b_transport, i);
                tSocket[i].register_nb_transport_fw(this,
                    &BUS::nb_transport_fw, i);
            }

            for(unsigned int i = 0; i < I; i++)
            {
                iSocket[i].register_nb_transport_bw(this,
                    &BUS::nb_transport_bw, i);
            }
        }

        private:
            std::map<tlm::tlm_generic_payload*, int> bwRoutingTable;
            std::map<tlm::tlm_generic_payload*, int> fwRoutingTable;

            virtual void b_transport( int id,
                                     tlm::tlm_generic_payload& trans,
                                     sc_time& delay )
            {
                {
                    sc_assert(id < T);
                    int outPort = routeFW(id, trans, false);
                    iSocket[outPort]->b_transport(trans, delay);
                }
            }
        };
};
```

```
virtual tlm::tlm_sync_enum nb_transport_fw( int id,
                                             tlm::tlm_generic_payload& trans,
                                             tlm::tlm_phase& phase,
                                             sc_time& delay )
{
    sc_assert(id < T);
    int outPort = 0;

    if(phase == tlm::BEGIN_REQ) {
        trans.acquire();
        outPort = routeFW(id, trans, true);
    }
    else if(phase == tlm::END_RESP) {
        outPort = fwRoutingTable[&trans];
        trans.release();
    }
    else {
        SC_REPORT_FATAL(name(), "ERROR!");
    }

    return iSocket[outPort]->nb_transport_fw(
        trans, phase, delay);
}

virtual tlm::tlm_sync_enum nb_transport_bw( int id,
                                             tlm::tlm_generic_payload& trans,
                                             tlm::tlm_phase& phase,
                                             sc_time& delay )
{
    int inPort = bwRoutingTable[&trans];

    return tSocket[inPort]->nb_transport_bw(
        trans, phase, delay);
}
};
```

# Simple Sockets: Target Example

```
int routeFW(int inPort, tlm::tlm_generic_payload &trans, bool store)
{
    int outPort = 0;

    if(trans.get_address() < 512)
    {
        outPort = 0;
    }
    else if(trans.get_address() >= 512 && trans.get_address() < 1024)
    {
        trans.set_address(trans.get_address() - 512);
        outPort = 1;
    }
    else {
        trans.set_response_status( tlm::TLM_ADDRESS_ERROR_RESPONSE );
    }

    if(store) {
        bwRoutingTable[&trans] = inPort;
        fwRoutingTable[&trans] = outPort;
    }

    return outPort;
}
```

Implementation of  
the memory map

Correct address,  
i.e. subtract offset  
such that target  
gets always  
addresses starting  
at 0

Store from where  
transaction comes  
and where it goes

# Simple Sockets: Target Example

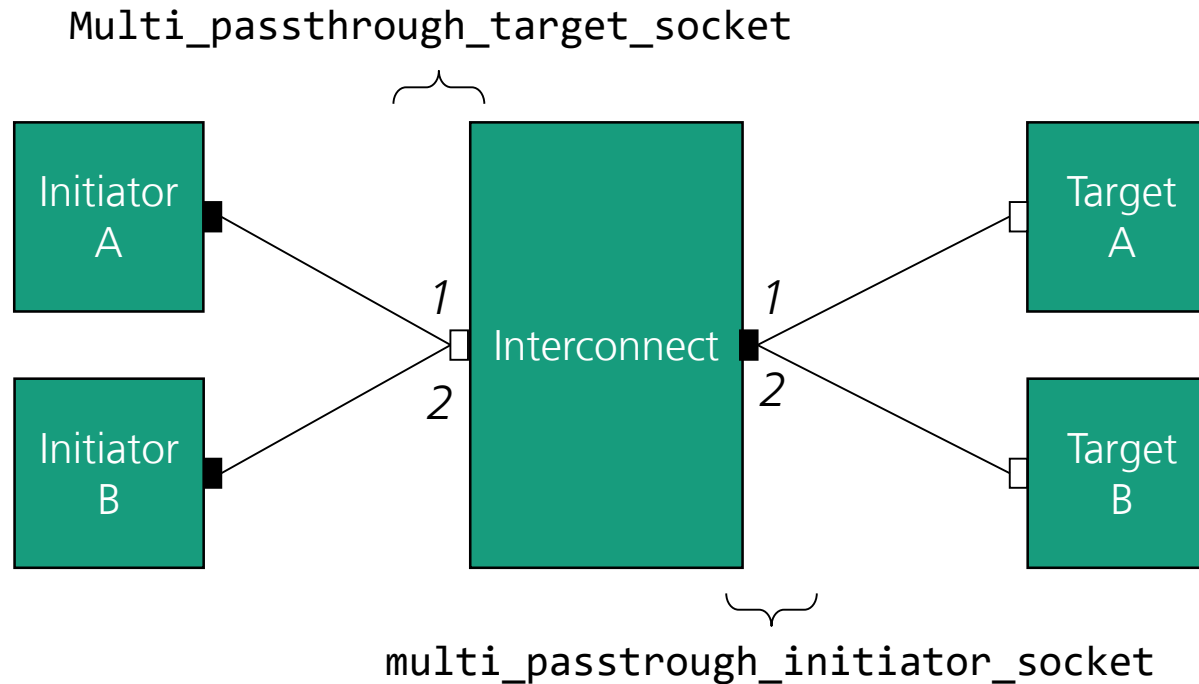
```
int sc_main (...)  
{  
  
    Initiator * cpu1    = new Initiator("C1");  
    Initiator * cpu2    = new Initiator("C2");  
  
    Target * memory1    = new Target("M1");  
    Target * memory2    = new Target("M2");  
  
    Interconnect<2,2> * bus = new BUS<2,2>("B1");  
  
    cpu1->iSocket.bind(bus->tSocket[0]);  
    cpu2->iSocket.bind(bus->tSocket[1]);  
  
    bus->iSocket[0].bind(memory1->tSocket);  
    bus->iSocket[1].bind(memory2->tSocket);  
  
    sc_start();  
  
    return 0;  
}
```

Number of components must be known at compile time!

Try code on github:

[https://github.com/TUK-SCVP/SCVP.artifacts/blob/master/tlm\\_simple\\_sockets/](https://github.com/TUK-SCVP/SCVP.artifacts/blob/master/tlm_simple_sockets/)

# Multipassthrough Sockets



- Similar to Tagged sockets: also an id is used for identification
- The id is determined by the order of the binding to the multipassthrough socket
- Dynamic binding, number of comp. does not have to be known at compile time

# Multipassthrough Sockets: Interconnect Example

```
SC_MODULE(BUS)
{
    public:
        tlm_utils::multi_passthrough_target_socket<BUS> tSocket;
        tlm_utils::multi_passthrough_initiator_socket<BUS> iSocket;

        SC_CTOR(Interconnect) : tSocket("tSocket"), iSocket("iSocket")
        {
            tSocket.register_b_transport(this, &BUS::b_transport);
            tSocket.register_nb_transport_fw(this, &BUS::nb_transport_fw);
            iSocket.register_nb_transport_bw(this, &BUS::nb_transport_bw);
        }

        private:
            std::map<tlm::tlm_generic_payload*, int> bwRoutingTable;
            std::map<tlm::tlm_generic_payload*, int> fwRoutingTable;

            void b_transport( int id, tlm::tlm_generic_payload& trans, sc_time& delay ) {
                ...
            }

            tlm::tlm_sync_enum nb_transport_fw( int id, ... ) {
                ...
            }

            virtual tlm::tlm_sync_enum nb_transport_bw( int id, ... ) {
                ...
            }
};
```

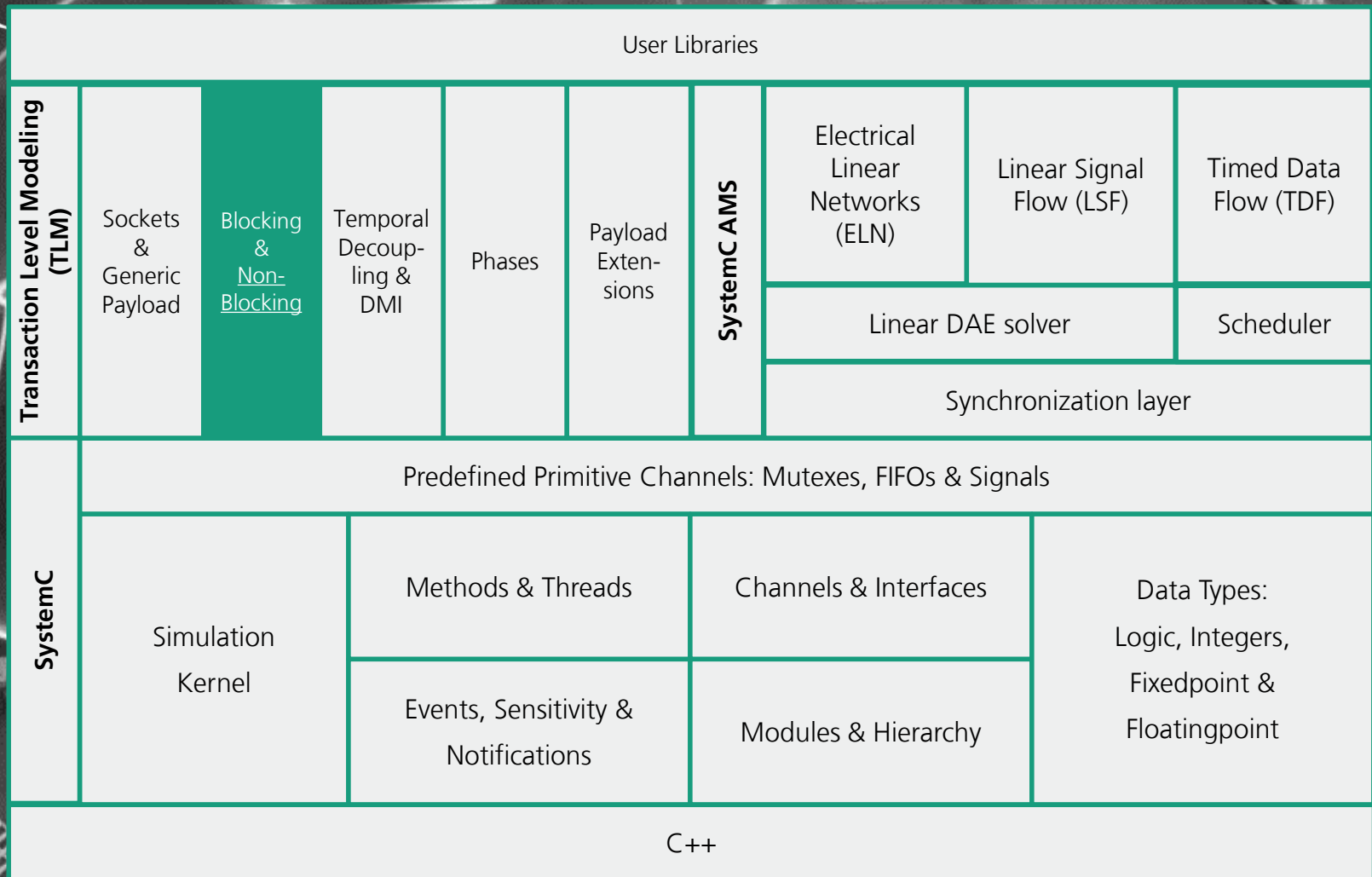
# Multipasstrough Sockets: Interconnect Example

```
int sc_main (...)  
{  
    Initiator * cpu1    = new Initiator("C1");  
    Initiator * cpu2    = new Initiator("C2");  
  
    Target * memory1    = new Target("M1");  
    Target * memory2    = new Target("M2");  
  
    Interconnect * bus = new BUS("B1");  
  
    cpu1->iSocket.bind(bus->tSocket);  
    cpu2->iSocket.bind(bus->tSocket);  
  
    bus->iSocket.bind(memory1->tSocket);  
    bus->iSocket.bind(memory2->tSocket);  
  
    sc_start();  
  
    return 0;  
}
```

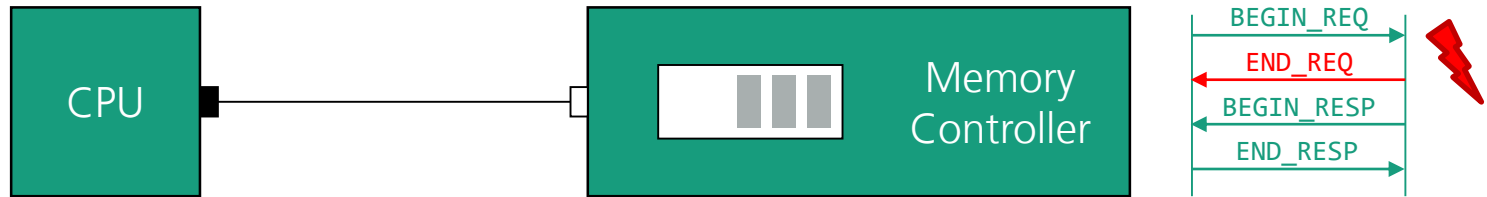
The order of the binding determines the ids for the function calls!

Try code on github:

[https://github.com/TUK-SCVP/SCVP.artifacts/blob/master/tlm\\_multipasstrough\\_sockets](https://github.com/TUK-SCVP/SCVP.artifacts/blob/master/tlm_multipasstrough_sockets)



# Modelling of Backpressure



- The exclusion rules enable flow control like back pressure:
  - E.g. if an input buffer of a target is full the target can defer the sending of **END\_REQ** for the transaction that filled the buffer, until the buffer has available space again
  - An RTL ready signal can be modeled by deferring **END\_REQ**
  - However, since it is non-blocking, the target can do something else, e.g. sending
- Also Response exclusion rule must be honored!



# Modelling of Backpressure

HW

```
DECLARE_EXTENDED_PHASE(INTERNAL);
```

```
SC_MODULE(Target) {
```

```
    bool responseInProgress;  
    tlm::tlm_generic_payload* endRequestPending;  
    tlm_utils::peq_with_cb_and_phase<Target> peq;  
    unsigned int numberOfTransactions;  
    unsigned int bufferSize;  
    std::queue<tlm::tlm_generic_payload*> responseQueue;  
    ...
```

```
    void peqCallback(...) {  
        sc_time delay;
```

```
        if(phase == tlm::BEGIN_REQ) {  
            trans.acquire();  
            if (numberOfTransactions < bufferSize) {  
                sendEndRequest(trans);  
            } else {  
                endRequestPending = &trans;  
            }  
        }
```

```
    } else if (phase == tlm::END_RESP) {  
        ...  
        numberOfTransactions--;  
        ...  
        if (responseQueue.size() > 0) {  
            gp * next = responseQueue.front();  
            responseQueue.pop();  
            sendResponse(*next);  
        }  
        if (endRequestPending) {  
            sendEndRequest(*endRequestPending);  
            endRequestPending = 0;  
        }  
    }
```

```
    else if(phase == INTERNAL) {  
        executeTransaction(trans);  
        if (responseInProgress) {  
            responseQueue.push(&trans);  
        } else {  
            sendResponse(trans);  
        }  
    }  
}
```

Internal protocol phase, instead of using a sensitive process!

Check if input buffer is full

Reduce transaction counter

Send next response

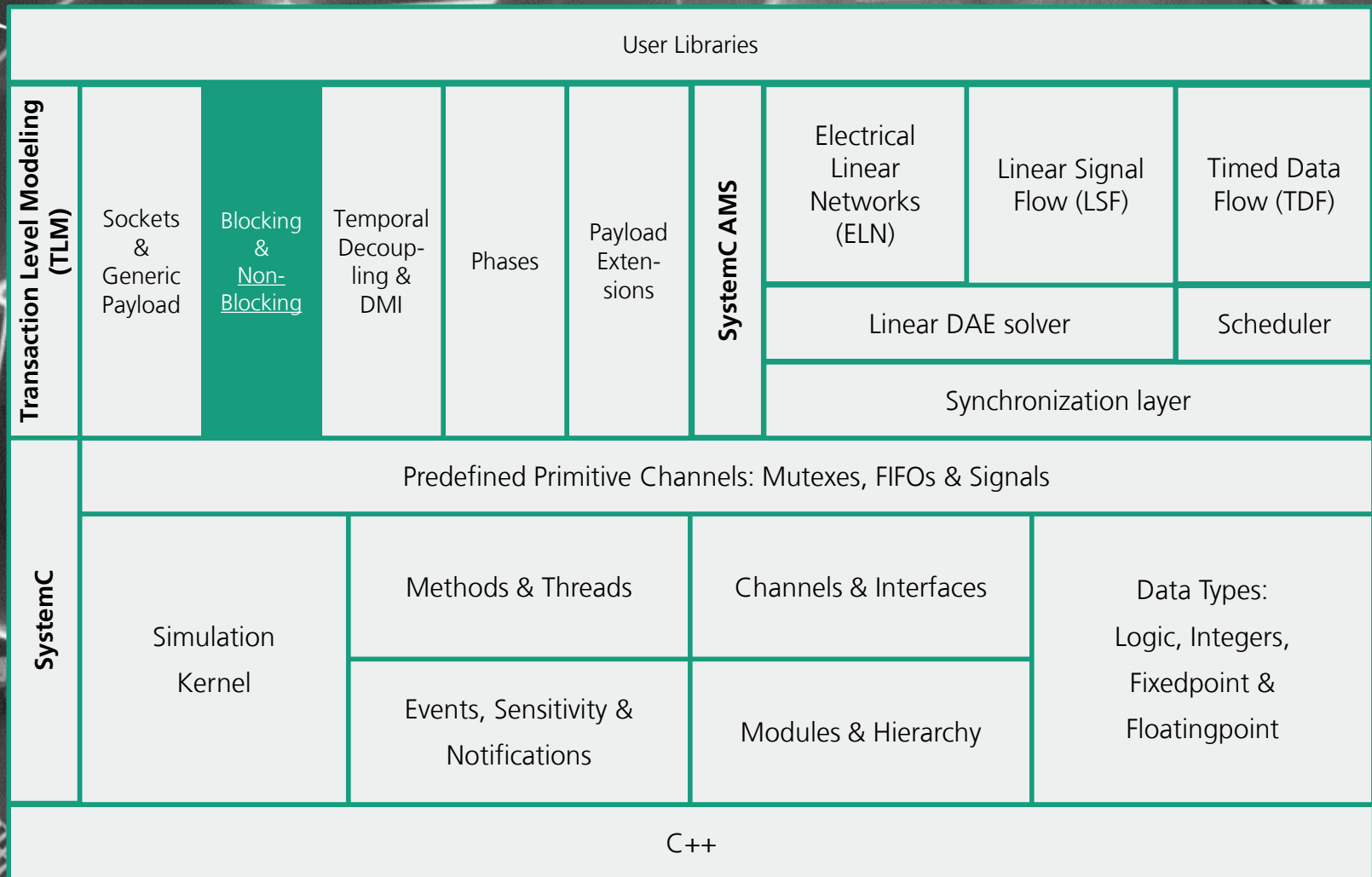
Unblock Initiator

Honor response exclusion rule

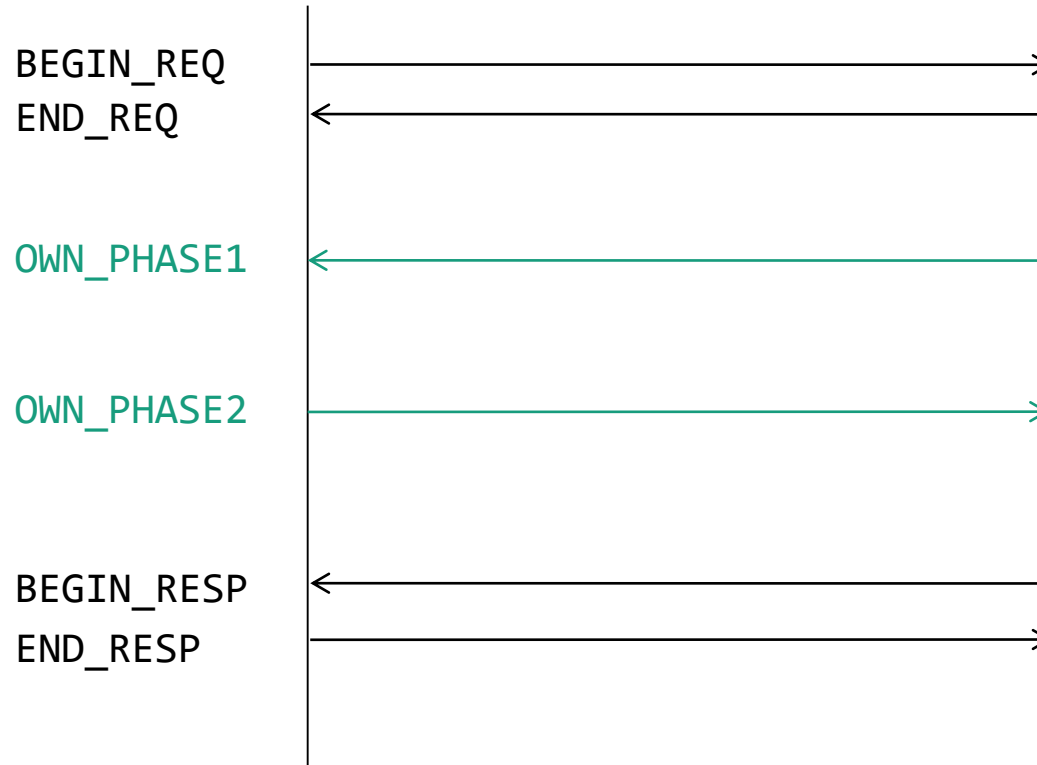
```
void sendEndRequest(tlm::tlm_generic_payload& trans) {  
    tlm::tlm_phase bw_phase;  
    sc_time delay;  
    bw_phase = tlm::END_REQ;  
    delay = randomDelay();  
  
    tlm::tlm_sync_enum status;  
    status = socket->nb_transport_bw(trans, bw_phase, delay);  
    delay = delay + randomDelay();  
    peq.notify(trans, INTERNAL, delay);  
  
    numberOfTransactions++;  
    printBuffer(bufferSize, numberOfTransactions);  
}  
  
void sendResponse(tlm::tlm_generic_payload& trans) {  
    tlm::tlm_sync_enum status;  
    tlm::tlm_phase bw_phase;  
    sc_time delay;  
  
    responseInProgress = true;  
    bw_phase = tlm::BEGIN_RESP;  
    delay = SC_ZERO_TIME;  
    status = socket->nb_transport_bw(trans, bw_phase, delay);  
  
    if (status == tlm::TLM_UPDATED) {  
        peq.notify(trans, bw_phase, delay);  
    }  
    else if (status == tlm::TLM_COMPLETED) {  
        numberOfTransactions--;  
        responseInProgress = false;  
    }  
    trans.release();  
}  
};
```

Try code on github:

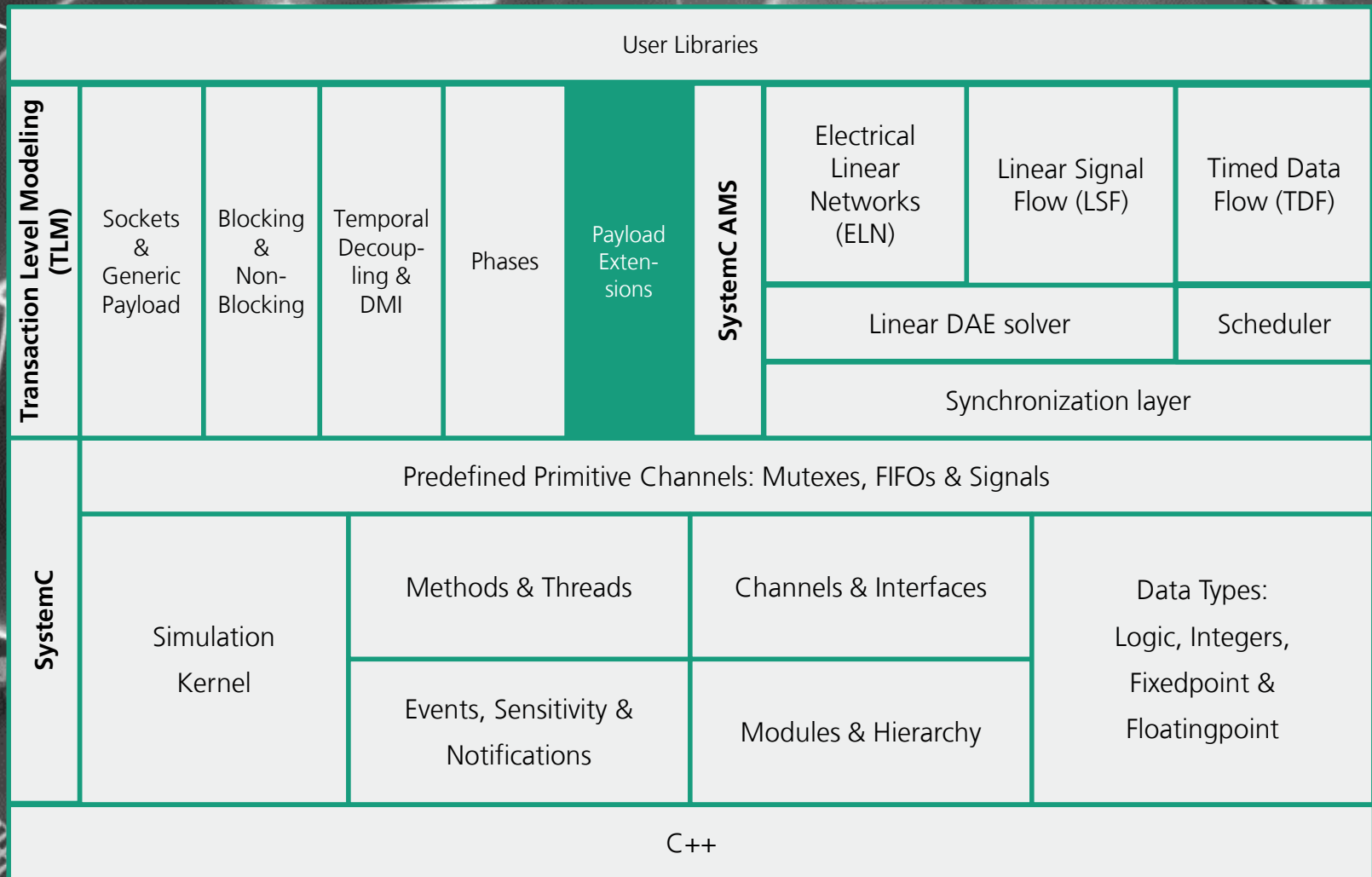
[https://github.com/TUK-SCVP/SCVP.artifacts/blob/master/tlm\\_at\\_backpressure](https://github.com/TUK-SCVP/SCVP.artifacts/blob/master/tlm_at_backpressure)



# Self defined Protocol Phases



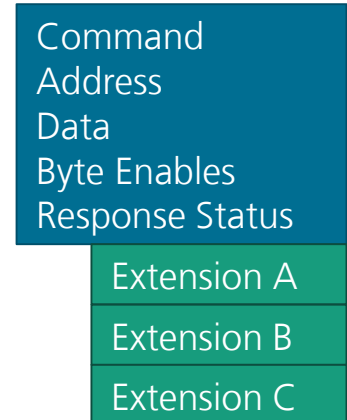
- Base protocol can be enhanced by phase extensions
- Usage of the macro: **DECLARE\_EXTENDED\_PHASE**(INTERNAL);
- More synchronization points → higher accuracy → slower simulation speed



# Payload Extensions

- Payload extensions are a flexible and powerful method to carry additional non-standard attributes (e.g. priority, ...)
- Extensions are attached to the normal generic payload
- Extensions can be:
  - Ignorable (ensures compatibility with base protocol)
  - Mandatory (a new protocol is needed!)
  - Private (only used by a single module e.g. Interconnect for storing routing information instead using maps)
  - Sticky (remain when transaction is returned to a pool)
  - Auto (freed when transaction is returned to a pool)

Generic payload object



# Payload Extensions Example

```
class routingExtension : public
tlm::tlm_extension<routingExtension> {
private:
    unsigned int inputPortNumber;
    unsigned int outputPortNumber;


public:
    routingExtension(unsigned int i, unsigned int o) :
        inputPortNumber(i), outputPortNumber(o)
    {}

    tlm_extension_base* clone() const {
        return new routingExtension(
            inputPortNumber, outputPortNumber);
    }

    void copy_from(const tlm_extension_base& ext) {
        const routingExtension& cpyFrom =
            static_cast<const routingExtension&>(ext);
        inputPortNumber = cpyFrom.getInputPortNumber();
        outputPortNumber = cpyFrom.getOutputPortNumber();
    }

    unsigned int getInputPortNumber() const {
        return inputPortNumber;
    }

    unsigned int getOutputPortNumber() const {
        return outputPortNumber;
    }
};
```



```
SC_MODULE(Interconnect)
{
    ...
    routingExtension* ext;

    ext = new routingExtension(
        inPort, outPort);

    trans.set_auto_extension(ext);
    ...
    routingExtension *ext = NULL;
    trans.get_extension(ext);
    outPort = ext->getOutputPortNumber();
    ...
};
```

Try code on github:  
[https://github.com/TUK-SCVP/SCVP.artifacts/blob/master/tlm\\_payload\\_extensions/](https://github.com/TUK-SCVP/SCVP.artifacts/blob/master/tlm_payload_extensions/)

