Heterogeneous Active Messages (HAM) — Implementing Lightweight Remote Procedure Calls in C++

Matthias Noack
noack@zib.de

Zuse Institute Berlin
Distributed Algorithms and Supercomputing
Context

You may remember me from such events as SC, ISC, IPDPS, or IXPUG ...
What I do at ZIB:

- **HPC-related** computer science research
  - programming models
  - performance and portability
- development of scientific codes
- user training/consulting for the HLRN supercomputer
- evaluation of upcoming HPC technologies
Context

What I do at ZIB:

• **HPC-related** computer science research
  • programming models
  • performance and portability

• development of scientific codes

• user training/consulting for the HLRN supercomputer

• evaluation of upcoming HPC technologies

Audience Survey

• Who is working in HPC?
• Who is familiar with RPCs/RMIs?
• Who is familiar with active messages?
Context

What I do at ZIB:

- **HPC-related** computer science research
  - programming models
  - performance and portability
- development of scientific codes
- user training/consulting for the HLRN supercomputer
- evaluation of upcoming HPC technologies

Audience Survey

- Who is working in HPC?
  - Who is familiar with RPCs/RMIs?
  - Who is familiar with active messages?
Context

What I do at ZIB:

• **HPC-related** computer science research
  • programming models
  • performance and portability
• development of scientific codes
• user training/consulting for the HLRN supercomputer
• evaluation of upcoming HPC technologies

Audience Survey

• Who is working in HPC?
• Who is familiar with RPCs/RMIs?
• Who is familiar with active messages?

You may remember me from such events as SC, ISC, IPDPS, or IXPUG ...
Context

What I do at ZIB:

• **HPC-related** computer science research
  • programming models
  • performance and portability

• development of scientific codes

• user training/consulting for the HLRN supercomputer

• evaluation of upcoming HPC technologies

Audience Survey

• Who is working in HPC?
• Who is familiar with RPCs/RMIs?
• Who is familiar with active messages?
What is this talk about?

Problem:

• Find the most light-weight, pure C++ implementation to do Remote Procedure Calls (RPCs) between possibly distributed and heterogeneous processes in an HPC context.
• Processes run executables from the same source
• Processes spawn and die together
• We do not require versioning, security, etc., and do not want the complexity of Interface Definition Languages (IDLs) and code generators.

Why?

• Foundation for an efficient and flexible C++ offloading framework.
• Target all architectures that can run a process and communicate somehow over an accessible API.
• Includes CPUs, Xeon Phi accelerators, NEC Vector Engine, ...
• Excludes direct support for current GPUs
What is this talk about?

Problem:

- Find the most light-weight, pure C++ implementation to do Remote Procedure Calls (RPCs) between possibly distributed and heterogeneous processes in an HPC context.
What is this talk about?

Problem:

- Find the most **light-weight, pure C++** implementation to do Remote Procedure Calls (**RPCs**) between possibly **distributed** and **heterogeneous** processes in an **HPC** context.
  - processes run executables from the same source
  - processes spawn and die together
What is this talk about?

Problem:

• Find the most light-weight, pure C++ implementation to do Remote Procedure Calls (RPCs) between possibly distributed and heterogeneous processes in an HPC context.
  • processes run executables from the same source
  • processes spawn and die together
• We do not require versioning, security, etc., and do not want the complexity of Interface Definition Languages (IDLs) and code generators.
What is this talk about?

Problem:

- Find the most **light-weight, pure C++** implementation to do Remote Procedure Calls (**RPCs**) between possibly **distributed** and **heterogeneous** processes in an **HPC** context.
  - processes run executables from the same source
  - processes spawn and die together
- We do **not require** versioning, security, etc., and do **not want** the complexity of Interface Definition Languages (IDLs) and code generators.

Why?
What is this talk about?

Problem:

• Find the most light-weight, pure C++ implementation to do Remote Procedure Calls (RPCs) between possibly distributed and heterogeneous processes in an HPC context.
  • processes run executables from the same source
  • processes spawn and die together

• We do not require versioning, security, etc., and do not want the complexity of Interface Definition Languages (IDLs) and code generators.

Why?

• Foundation for an efficient and flexible C++ offloading framework.
What is this talk about?

Problem:

- Find the most light-weight, pure C++ implementation to do Remote Procedure Calls (RPCs) between possibly distributed and heterogeneous processes in an HPC context.
  - processes run executables from the same source
  - processes spawn and die together

- We do not require versioning, security, etc., and do not want the complexity of Interface Definition Languages (IDLs) and code generators.

Why?

- Foundation for an efficient and flexible C++ offloading framework.
- Target all architectures that can run a process and communicate somehow over an accessible API.
What is this talk about?

Problem:

- Find the most **light-weight**, **pure C++** implementation to do Remote Procedure Calls (RPCs) between possibly **distributed** and **heterogeneous** processes in an **HPC** context.
  - processes run executables from the same source
  - processes spawn and die together
- We do **not require** versioning, security, etc., and do **not want** the complexity of Interface Definition Languages (IDLs) and code generators.

Why?

- Foundation for an efficient and flexible C++ offloading framework.
- Target all architectures that **can run a process and communicate** somehow over an accessible API.
  - includes CPUs, Xeon Phi accelerators, NEC Vector Engine, ...
User Perspective

What we want:

- execute some function in the address space of a remote process

```c
int fun(int a, int b) {
    return a + b;
}
```
User Perspective

What we want:

• execute some function in the address space of a remote process

```cpp
int fun(int a, int b) {
    return a + b;
}
```

• with something as close as possible to `std::async`:

```cpp
int main() {
    int a, b;  // init somehow

    // run asynchronously
    auto res_future =
        std::async(fun, a, b);
    int c = res_future.get();
}
```
User Perspective

What we want:

• execute some function in the address space of a remote process

```c
int fun(int a, int b) {
    return a + b;
}
```

• for an RPC we need a target process, and some kind of closure to transfer:

```c
int main() {
    int a, b; // init somehow
    node_t target; // target process

    // offload asynchronously
    auto res_future = // f2f() generates a closure
        offload::async(target, f2f(&fun, a, b));
    int c = res_future.get();
}
```
Active Messages and Heterogeneity

The most simple RPC implementation:

1. Use identical binaries for each process
2. Send an active message containing a function pointer
3. Call the function at the receiver

⇒ Only works if processes are homogeneous
   • Fails as soon as different binaries are generated
   • Due to different architectures, compilers, options, ...

Heterogeneous Active Messages (HAM):

• Enable a similar approach for differing, i.e. heterogeneous binaries
• E.g. by an efficient addresses translation mechanism
Active Messages and Heterogeneity

The most simple RPC implementation:

0. use **identical binaries** for each process
1. send an **active message** containing a function pointer
2. call the function at the receiver
Active Messages and Heterogeneity

The most simple RPC implementation:

0. use **identical binaries** for each process
1. send an **active message** containing a function pointer
2. call the function at the receiver

⇒ only works if processes are homogeneous

- fails as soon as different binaries are generated
- due to different architectures, compilers, options, ...
Active Messages and Heterogeneity

The most simple RPC implementation:

0. use \textbf{identical binaries} for each process
1. send an \textbf{active message} containing a function pointer
2. call the function at the receiver

$\Rightarrow$ only works if processes are homogeneous

- fails as soon as different binaries are generated
- due to different architectures, compilers, options, …

Heterogeneous Active Messages (HAM):
Active Messages and Heterogeneity

The most simple RPC implementation:

0. use **identical binaries** for each process
1. send an **active message** containing a function pointer
2. call the function at the receiver
⇒ only works if processes are homogeneous
   • fails as soon as different binaries are generated
   • due to different architectures, compilers, options, …

Heterogeneous Active Messages (HAM):

• enable a similar approach for differing, i.e. **heterogeneous binaries**
• e.g. by an efficient **addresses translation** mechanism
HAM (Heterogeneous Active Messages) and HAM-Offload
HAM (Heterogeneous Active Messages) and HAM-Offload

Problem: the RPC mechanism
a) kernel code deployment
b) efficient kernel invocation

Solution:

a) symmetric execution model:
   • build heterogeneous binaries from same source
b) **Heterogeneous Active Messages**
   • provide code address translation between heterogeneous processes in $O(1)$
   • use the C++ type-system to:
     • generate message handlers
     • build translation data structures
HAM (Heterogeneous Active Messages) and HAM-Offload

Problem:
- generic means to transfer Heterogeneous Active Messages and data

Solution:
- an abstract **Communication Backend**
- direct data transfers between offload targets
- implemented for different technologies

<table>
<thead>
<tr>
<th>HAM</th>
<th>Comm. Backend</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI</td>
<td>TCP/IP</td>
</tr>
<tr>
<td></td>
<td>Intel SCIF</td>
</tr>
<tr>
<td></td>
<td>NEC VEO/DMA</td>
</tr>
</tbody>
</table>
HAM (Heterogeneous Active Messages) and HAM-Offload

Problem:
- unified API for intra- and inter-node offloading

Solution:
- **HAM-Offload** C++ API
- offload primitives built on top of HAM and the communication back-end
- light-weight runtime for message execution
- similar functionality as vendor solutions
HAM-Offload Performance

Cost for offloading an empty kernel, i.e. the minimal overhead:

**Offload Cost: HAM–Offload vs. Vendor–Provided Solutions**

- **Intel Xeon Phi 5110P**
  - HAM–Offload: 1.8 µs
  - Vendor–Solution: 51.8 µs

- **NEC VE, Type 10B**
  - HAM–Offload: 6.1 µs
  - Vendor–Solution: 79.4 µs

⇒ While being language-only and high-level
HAM-Offload Performance

Cost for offloading an empty kernel, i.e. the minimal overhead:

![Offload Cost: HAM-Offload vs. Vendor-Provided Solutions](chart)

- **Intel Xeon Phi 5110P**
  - HAM-Offload: 1.8 µs
  - Vendor-Solution: 51.8 µs

- **NEC VE, Type 10B**
  - HAM-Offload: 6.1 µs
  - Vendor-Solution: 79.4 µs

... vs. **Intel LEO** (pragma-based compiler extension)
- **28.6x** speed-up, i.e. **96.5%** overhead reduction

... vs. **NEC VEO** (low-level C-API)
- **13.1x** speed-up, i.e. **92.3%** overhead reduction

⇒ while being language-only and high-level
HAM-Offload Performance

Cost for offloading an empty kernel, i.e. the minimal overhead:

Offload Cost: HAM-Offload vs. Vendor-Provided Solutions

<table>
<thead>
<tr>
<th>Platform</th>
<th>HAM-Offload</th>
<th>Vendor-Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEC VE, Type 10B</td>
<td>6.1 µs</td>
<td>79.4 µs</td>
</tr>
<tr>
<td>Intel Xeon Phi 5110P</td>
<td>1.8 µs</td>
<td>51.8 µs</td>
</tr>
</tbody>
</table>

... vs. Intel LEO (pragma-based compiler extension)
- $28.6 \times$ speed-up, i.e. $96.5\%$ overhead reduction

... vs. NEC VEO (low-level C-API)
- $13.1 \times$ speed-up, i.e. $92.3\%$ overhead reduction
HAM-Offload Performance

Cost for offloading an empty kernel, i.e. the minimal overhead:

**Offload Cost: HAM–Offload vs. Vendor–Provided Solutions**

<table>
<thead>
<tr>
<th>Processor</th>
<th>HAM–Offload</th>
<th>Vendor–Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon Phi 5110P</td>
<td>1.8 µs</td>
<td>51.8 µs</td>
</tr>
<tr>
<td>NEC VE, Type 10B</td>
<td>6.1 µs</td>
<td>79.4 µs</td>
</tr>
</tbody>
</table>

... vs. Intel LEO (pragma-based compiler extension)
- **28.6×** speed-up, i.e. **96.5 %** overhead reduction

... vs. NEC VEO (low-level C-API)
- **13.1×** speed-up, i.e. **92.3 %** overhead reduction

⇒ while being **language-only** and **high-level**
HAM (Heterogeneous Active Messages) and HAM-Offload

Problem: the RPC mechanism

a) kernel code deployment
b) efficient kernel invocation

Solution:

a) symmetric execution model:
   • build heterogeneous binaries from same source
b) Heterogeneous Active Messages
   • provide code address translation between heterogeneous processes in $O(1)$
   • use the C++ type-system to:
     • generate message handlers
     • build translation data structures
HAM Structure

```
active_msg_base
  void operator(void* msg)
  key_t handler_key

msg_handler_registry
  handler_t get_handler(key_t key)
  map: key_t → handler_t

function
  void operator()
  tuple<migratable<Pars>...> args;

execution_policy
  static void handler(void* msg)

Derived

active_msg
  static key_t handler_key_static

Derived, Policy

offload_msg
  void operator()
```

```
HAM RPC at Runtime

```cpp
// offload asynchronously
auto res_future =
    ham::async(target, f2f(&fun, a, b));
```
/ **HAM RPC at Runtime**

```cpp
// offload asynchronously
auto res_future =
    ham::async(target, f2f(&fun, a, b));
```

---

**Diagram:**

- Function + args
- f2f
- `ham::function`
- `ham::async`
- offload_msg
- send
- receive buffer
- reinterpret_cast<>()
- offload_msg handler
- operator()
- result
- active_msg_base
HAM Structure

**function functor**
- generated by f2f
- function **signature** as template **type** parameter
- function **address** as template **value** parameter

**migratable wrapper**
- hooks for serialisation/deserialisation
- conversion ctor from T
- conversion operator to T
/ offload asynchronously
auto res_future =
    ham::async(target, f2f(&fun, a, b));
HAM Structure

**active_msg_base**
- void operator(void* msg)
- key_t handler_key

**msg_handler_registry**
- handler_t get_handler(key_t key)
- map: key_t → handler_t

**execution_policy**
- static void handler(void* msg)

**active_msg**
- static key_t handler_key_static

**function**
- void operator()
- tuple<migratable<Pars>...> args;

**offload_msg**
- inherits a function instantiation
- inherits from active_msg, passing its type upwards (CRTP)
- just an example of how HAM is used in HAM-Offload
HAM RPC at Runtime

```cpp
// offload asynchronously
auto res_future = 
    ham::async(target, f2f(&fun, a, b));
```

Receiving side:
- **typeless buffer**
- all messages inherit from `active_msg_base`
- can be called with the receive buffer
**HAM Structure**

- **active_msg_base**
  - trivial, callable base class
  - looks up its `handler_key` at the `msg_handler_registry` and calls it

- **msg_handler_registry**
  - `handler_t get_handler(key_t key)`
  - `map: key_t → handler_t`

- **active_msg**
  - `static key_t handler_key_static`

- **execution_policy**
  - `static void handler(void* msg)`

- **function**
  - `void operator()`
  - `tuple< migratable< Pars > ... > args;`

- **offload_msg**
  - `void operator()`
active_msg_base
- trivial, callable base class
- looks up its handler_key at the msg_handler_registry and calls it

msg_handler_registry
- LUT: handler key to local function address in O(1)
HAM Structure

**active_msg_base**
- trivial, callable base class
- looks up its `handler_key` at the `msg_handler_registry` and calls it

**msg_handler_registry**
- LUT: handler key to local function address in O(1)

**execution_policy**
- the actual handler
- **upcasts** to Derived

**active_msg**
- upcasts to `Derived`

**offload_msg**
- void operator()
The HAM (Handler Management) structure includes:

- The `active_msg_base` class with a `void operator(void* msg)` method and a `key_t handler_key`.
- The `msg_handler_registry` class with a `handler_t get_handler(key_t key)` method and a `map: key_t → handler_t`.
- The `function` class with a `void operator()` method and a `tuple<migratable<Pars>> args`.
- The `offload_msg` class with a `void operator()` method.
- The `execution_policy` class with a `static void handler(void* msg)` method.
- The `active_msg` class with a `static key_t handler_key_static`.

**active_msg**

- Links the message type to its handler key, i.e., O(1) look-up.
- Static member init. provides a hook for collecting handler addresses prior to main.

⇒ Collect addresses and `typeid().name()`.
HAM Address Translation

- keys are valid across binaries, addresses are not
- keys are defined by the lexicographical order of the message-type’s typeid names
  ⇒ coordination of global keys without communication
- requires compatible C++ ABIs across compilers (icc, clang, gcc, ncc) and platforms (x86, KNC/KNL, VE, ARM)
  ⇒ most ABIs refer to the IA-64 C++ ABI for the relevant parts
Handler Maps and C++ RTTI Names

BEGIN HANDLER MAP

typeid_name:
    N3ham3msg10active_msgINS_7offload6detail11offload_msgINS2_7runtime17terminate_functorENS0_23execution_policy_directEEES7_EE
handler_address: 0x440d10

typeid_name:
    N3ham3msg10active_msgINS_7offload6detail18offload_result_msgINS_8functionIPFiiEXadL_ZZ13ham_user_mainiPPcEN3$_08__invokeEiEEEENS0_24default_execution_policyEEESC_EE
handler_address: 0x42a7e0

typeid_name:
    N3ham3msg10active_msgINS_7offload6detail18offload_result_msgINS_8functionIPFvvEXadL_Z7fun_onevEEEENS0_24default_execution_policyEEES9_EE
handler_address: 0x42db20

END HANDLER MAP

index: 0, handler_address: 0x440d10
index: 1, handler_address: 0x42a7e0
index: 2, handler_address: 0x42db20
Functions, Functors, and Lambdas

The function template:

```cpp
// function signature as template type parameter
// function pointer as template value parameter
template <typename Result, typename ... Pars,
          Result (*)(Pars...)>
class function <Result (*)(Pars...), FunctionPtr> {
public:
    // variadic constructor template
    // takes compatible argument types
    template <typename ... Args>
    function(Args&&... arguments);

    Result operator()() const;

private:
    std::tuple<migratable<Pars>...> args;
};
```
HAM Structure

**Function Functors**
- generated by f2f
- function signature as template type parameter
- function address as template value parameter

**Migratable Wrapper**
- hooks for serialisation/deserialisation
- conversion ctor from T
- conversion operator to T
**Functions, Functors, and Lambdas**

The function template:

```cpp
// function signature as template type parameter
// function pointer as template value parameter
template<typename Result, typename... Pars,
        Result (*FunctionPtr)(Pars...)
class function<Result (*)(Pars...), FunctionPtr> {
public:
    // variadic constructor template
    // takes compatible argument types
    template<typename... Args>
    function(Args&&... arguments);

    Result operator()() const;

private:
    std::tuple<migratable<Pars>...> args;
};
```
Functions, Functors, and Lambdas

The function template:

```cpp
// function signature as template type parameter
// function pointer as template value parameter
template<typename Result, typename... Pars,
         Result (*FunctionPtr)(Pars...)>  
class function<Result (*)(Pars...), FunctionPtr> { ... }; 
```

Cumbersome instantiation:

```cpp
function<decltype(fun_ptr), fun_ptr>(/* arguments */);
```
Functions, Functors, and Lambdas

The function template:

```cpp
// function signature as template type parameter
// function pointer as template value parameter
template<typename Result, typename... Pars,
    Result (*FunctionPtr)(Pars...)>
class function<Result (*)(Pars...), FunctionPtr> { ... };
```

Cumbersome instantiation:

```cpp
function<decltype(fun_ptr), fun_ptr>(/* arguments */);
```

Hence the f2f (variadic macro):

```cpp
// f2f = "function to functor"
// NOTE: the '& is required
f2f(&fun, /* arguments */);
```
Functions, Functors, and Lambdas

The function template:

```cpp
// function signature as template type parameter
// function pointer as template value parameter
template<typename Result, typename... Pars,
    Result (*FunctionPtr)(Pars...)>
class function<Result (*)(Pars...), FunctionPtr> { ... };
```

Cumbersome instantiation:

```cpp
function<decltype(fun_ptr), fun_ptr>(/* arguments */);
```

Hence the f2f (with C++17):

```cpp
template<auto fun_ptr>
using f2f = function<decltype(fun_ptr), fun_ptr>;
// C++17 f2f syntax:
// NOTE: the '& before fun can be skipped
f2f<fun>(/* arguments */);
```
Functions, Functors, and Lambdas

So what about Lambdas?

- capturing lambdas are not tractable as their state is inaccessible
- **captureless** lambdas have an implicit conversion operator to function pointer, which is `constexpr` since C++17
  ⇒ can be used as template value argument
Functions, Functors, and Lambdas

So what about Lambdas?

- capturing lambdas are not tractable as their state is inaccessible
- **captureless** lambdas have an implicit conversion operator to function pointer, which is `constexpr` since C++17
  - can be used as template value argument

Requires a little convincing, though:

```cpp
// NOT possible, lambda used as template argument
f2f [](/* Parse */) { /* do something */ }>
  (/* args */);

// possible: unary + operator triggers
// conversion to function pointer
f2f <+[](/* Parse */) { /* do something */ }>
  (/* args */);
```
Functions, Functors, and Lambdas

So what about Lambdas?

- capturing lambdas are not tractable as their state is inaccessible
- **captureless** lambdas have an implicit conversion operator to function pointer, which is constexpr since C++17
  
  \[ \Rightarrow \] can be used as template value argument

The ’+’ can be somewhat hidden:

```cpp
// lambda to function (L as type argument)
template<typename L, typename Args...>
auto l2f(L lambda, Args&&... args) {
    // conversion to pointer through +
    return f2f <+ lambda >(std::forward<Args>(args)...);
}

// resulting syntax:
l2f([](/* Pars */){ /* do sth. */ }, /* args */);
```
Functions, Functors, and Lambdas

Final syntaxes:

```c++
// some offloaded function

template int square(int x) {
    return x * x;
}

// offload functor, f2f as macro (pre C++17)
offload::async(target, f2f(&square, 42));

// offload functor, f2f auto template (C++17)
offload::async(target, f2f<square>(42));

// offload anonymous lambda (C++17)
offload::async(target, l2f([](int x) { return x * x; }, 42));
```
Handler Maps and C++ RTTI Names

BEGIN HANDLER MAP

typeid_name:
N3ham3msg10active_msgINS_7offload6detail11offload_msgINS2_7runtime17terminate_functorENS0_23execution_policy_directEEES7_EE
handler_address: 0x440d10

typeid_name:
N3ham3msg10active_msgINS_7offload6detail18offload_result_msgINS_8functionIPFiiEXadL_ZZ13ham_user_mainiPPcEN3$_08__invokeEEEEENS0_24default_execution_policyEEESCE
handler_address: 0x42a7e0

typeid_name:
N3ham3msg10active_msgINS_7offload6detail18offload_result_msgINS_8functionIPFvvEXadL_Z7fun_onevEEEENS0_24default_execution_policyEEES9_EE
handler_address: 0x42db20

END HANDLER MAP

index: 0, handler_address: 0x440d10
index: 1, handler_address: 0x42a7e0
index: 2, handler_address: 0x42db20
Handler Maps and C++ RTTI Names

BEGIN HANDLER MAP

typeid_name:
N3ham3msg10active_msgINS_7offload6detail11offload_msgINS2_7runtime17terminate_functorENS0_23execution_policy_directEEES7_EE
handler_address: 0x440d10

typeid_name:
N3ham3msg10active_msgINS_7offload6detail18offload_result_msgINS_8functionIPFiiEXadL_ZZ13ham_user_mainiPPcEN3$_08__invokeEiEEEENS0_24default_execution_policyEEESC_EE
handler_address: 0x42a7e0

typeid_name:
N3ham3msg10active_msgINS_7offload6detail18offload_result_msgINS_8functionIPFvvEXadL_Z7fun_onevEEEENS0_24default_execution_policyEEES9_EE
handler_address: 0x42db20

END HANDLER MAP

index: 0, handler_address: 0x440d10
index: 1, handler_address: 0x42a7e0
index: 2, handler_address: 0x42db20
Summary

Implementing an RPC mechanism like HAM reveals three things when it comes to distributed and heterogeneous systems:

- **C++** is already capable of a lot, even without language support:
  - library solutions, template code generation, wrappers, smart-pointers, ...

- Limitations of the current standard:
  - Mostly implementation-defined, i.e. unstandardised aspects
    ⇒ review and reduce
  - ABI, RTTI, types like `long double`
    ⇒ ensure compiler interoperability of (new) features

- Seemingly incompatible features:
  - Complex, compiler-generated code, e.g. from lambda expressions
    ⇒ take distributed/heterogeneous systems into account
Summary

Implementing an RPC mechanism like HAM reveals three things when it comes to distributed and heterogeneous systems:

C++ is already capable of a lot, even without language support:

- library solutions, template code generation, wrappers, smart-pointers, …
Summary

Implementing an RPC mechanism like HAM reveals three things when it comes to distributed and heterogeneous systems:

C++ is already capable of a lot, even without language support:
- library solutions, template code generation, wrappers, smart-pointers, ...

Limitations of the current standard:
- mostly implementation-defined, i.e. unstandardised aspects
  ⇒ review and reduce
- ABI, RTTI, types like long double, ...
  ⇒ ensure compiler interoperability of (new) features
Summary

Implementing an RPC mechanism like HAM reveals three things when it comes to distributed and heterogeneous systems:

C++ is already capable of a lot, even without language support:

- library solutions, template code generation, wrappers, smart-pointers, ...

Limitations of the current standard:

- mostly implementation-defined, i.e. unstandardised aspects  
  \( \Rightarrow \) review and reduce
- ABI, RTTI, types like long\_double, ...
  \( \Rightarrow \) ensure compiler interoperability of (new) features

Seemingly incompatible features:

- complex, compiler-generated code, e.g. from lambda expressions
  \( \Rightarrow \) take distributed/heterogeneous systems into account
Thank you.

Feedback? Questions? Ideas?

noack@zib.de

https://github.com/noma/ham