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

Matthias Noack
noack@zib.de

Zuse Institute Berlin
Distributed Algorithms and Supercomputing

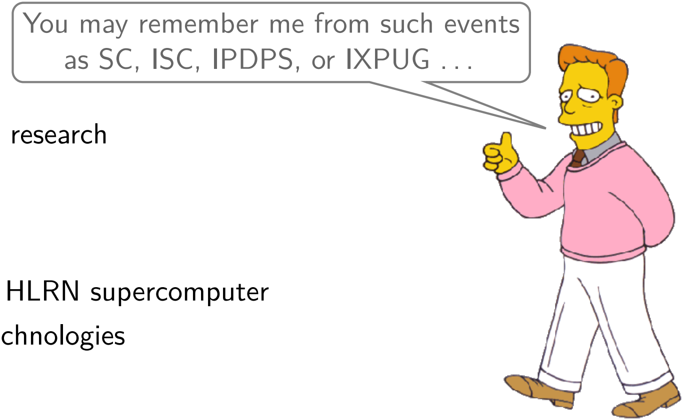
Context

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

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



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

Audience Survey

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

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

Audience Survey

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

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

Audience Survey

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

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

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:

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 accessibe 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 accessibe API.
  - includes CPUs, Xeon Phi accelerators, NEC Vector Engine, ...
  - excludes direct support for current GPUs

User Perspective

What we want:

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

```
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

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

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

```
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

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

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

```
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:

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 **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):

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

HAM

Problem: the RPC mechanism

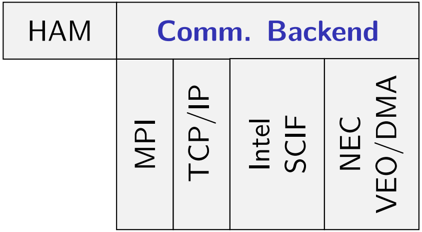
a) kernel code deployment
b) efficient kernel invocation

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



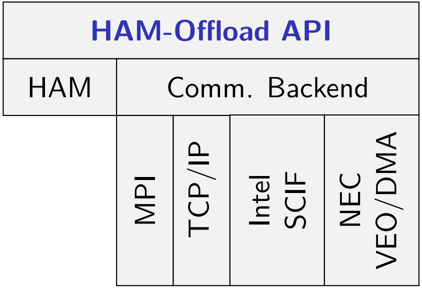
Problem:

- generic means to transfer Heterogeneous Active Messages and data

Solution:

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

HAM (Heterogeneous Active Messages) and HAM-Offload



Problem:

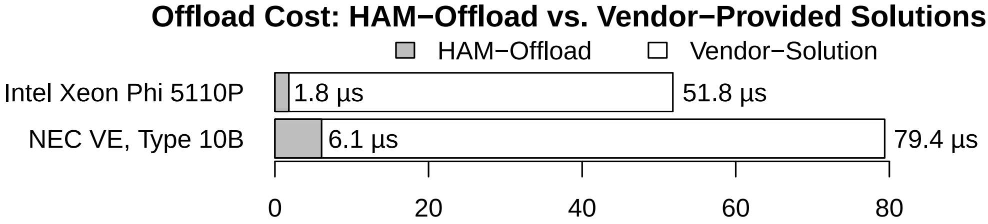
- unified API for intra- and inter-node offloading

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

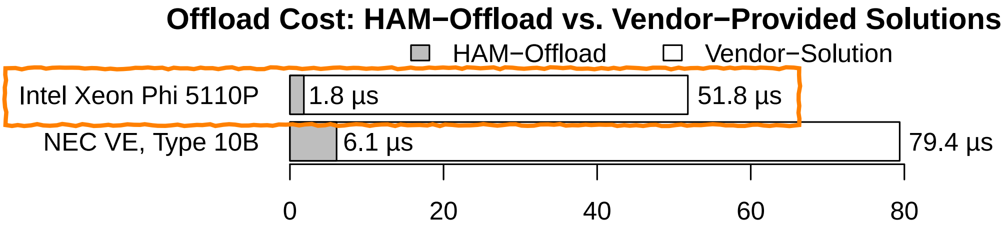
HAM-Offload Performance

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



HAM-Offload Performance

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

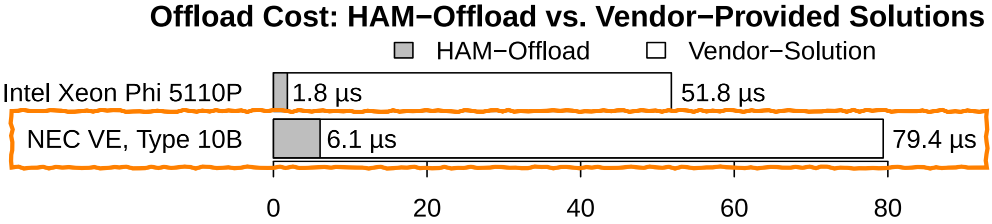


. . . **vs. Intel LEO** (pragma-based compiler extension)

- **28.6**× speed-up, i.e. **96.5 %** overhead reduction

HAM-Offload Performance

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



. . . **vs. Intel LEO** (pragma-based compiler extension)

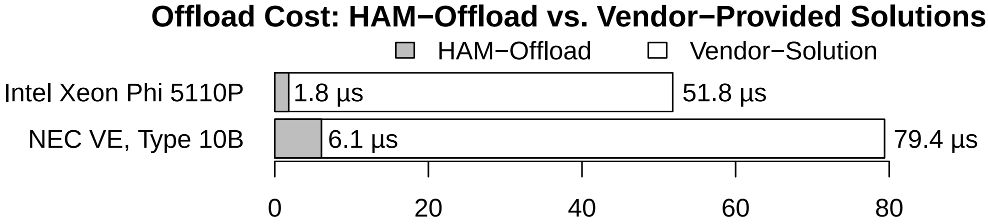
- **28.6**× speed-up, i.e. **96.5 %** overhead reduction

. . . **vs. NEC VEO** (low-level C-API)

- **13.1**× speed-up, i.e. **92.3 %** overhead reduction

HAM-Offload Performance

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



... **vs. Intel LEO** (pragma-based compiler extension)

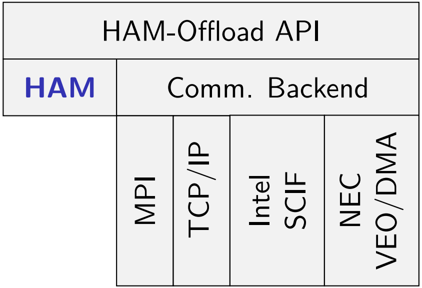
- **28.6**× speed-up, i.e. **96.5 %** overhead reduction

... **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

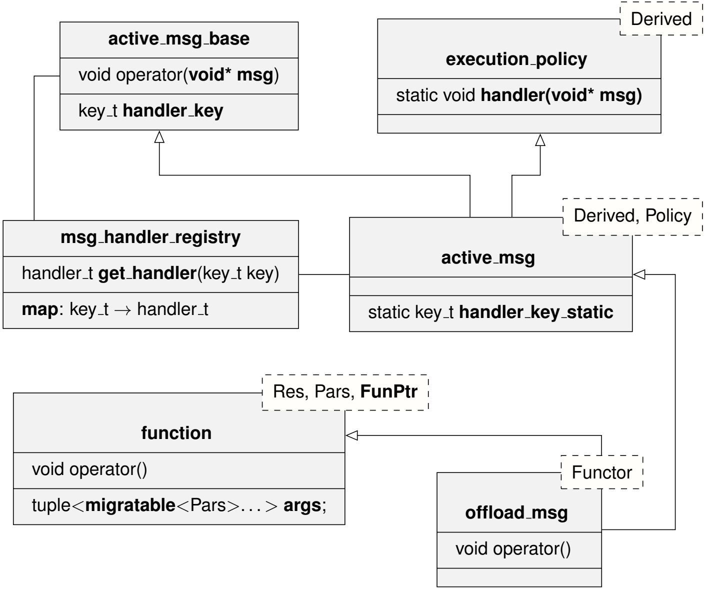
a) kernel code deployment
b) efficient kernel invocation

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**

Derived

execution_policy

static void **handler(void* msg)**

Derived, Policy

active_msg

static key_t **handler_key_static**

msg_handler_registry

handler_t **get_handler**(key_t key)

map: key_t → handler_t

Res, Pars, **FunPtr**

function

void operator()

tuple<**migratable**<Pars>. . . > **args**;

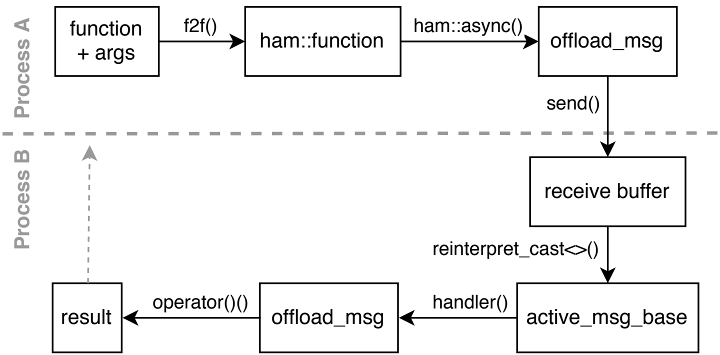
Functor

offload_msg

void operator()

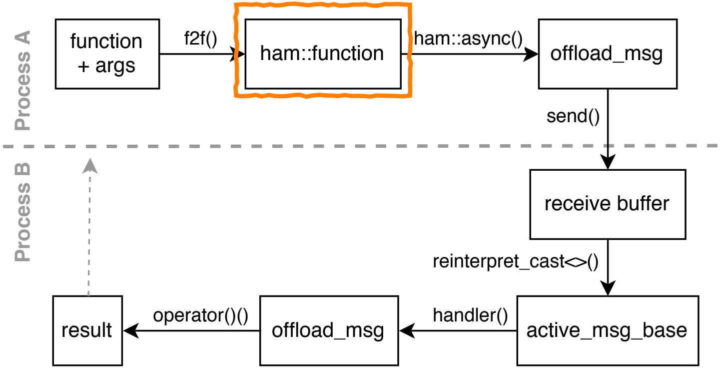
HAM RPC at Runtime

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

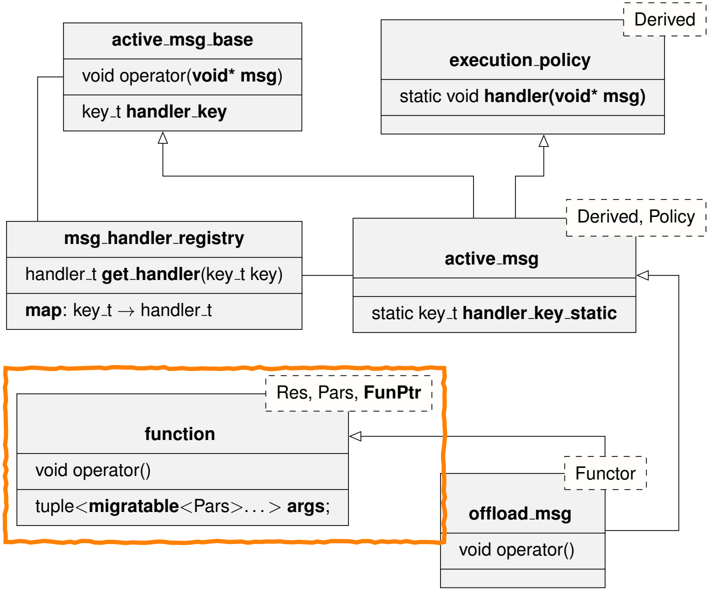


HAM RPC at Runtime

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



HAM Structure



`function functor`

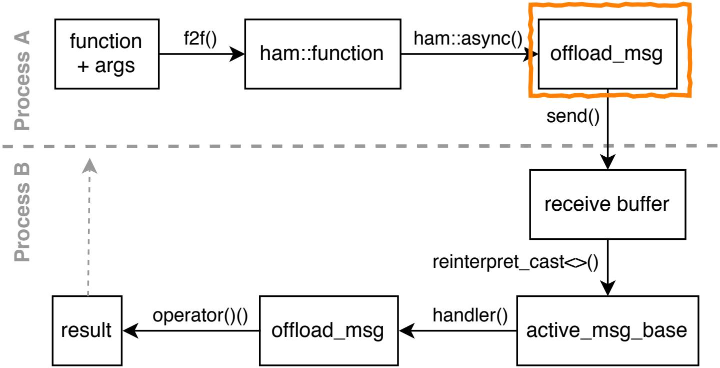
- generated by `f2f`
- function **signature** as template **type** parameter
- function **address** as template **value** parameter

`migratable wrapper`

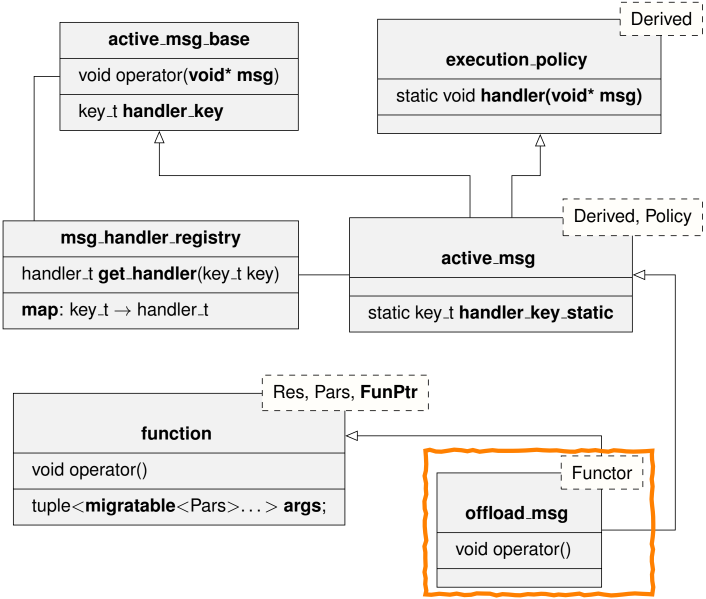
- hooks for serialisation/deserialisation
- conversion ctor from `T`
- conversion operator to `T`

HAM RPC at Runtime

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



HAM Structure

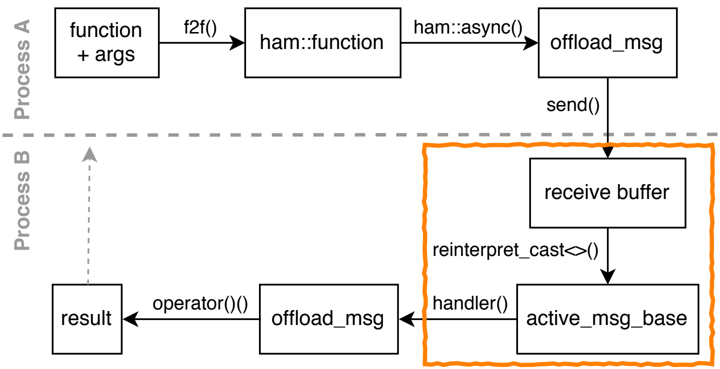


`offload_msg`

- 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

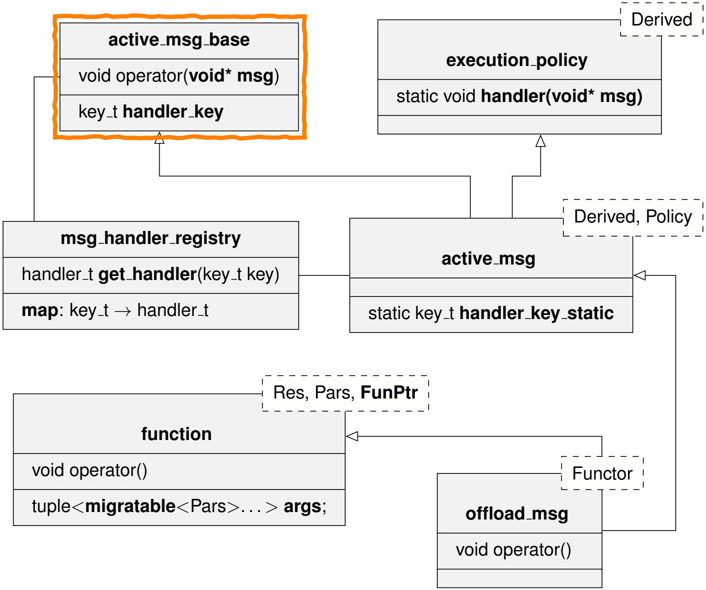
```
// offload asynchronously
auto res_future =
  ham::async(target, f2f(&fun, a, b));
```



Receving side:

- **typeless buffer**
- all messages inherit from `active_msg_base`
- can be called with the receive buffer

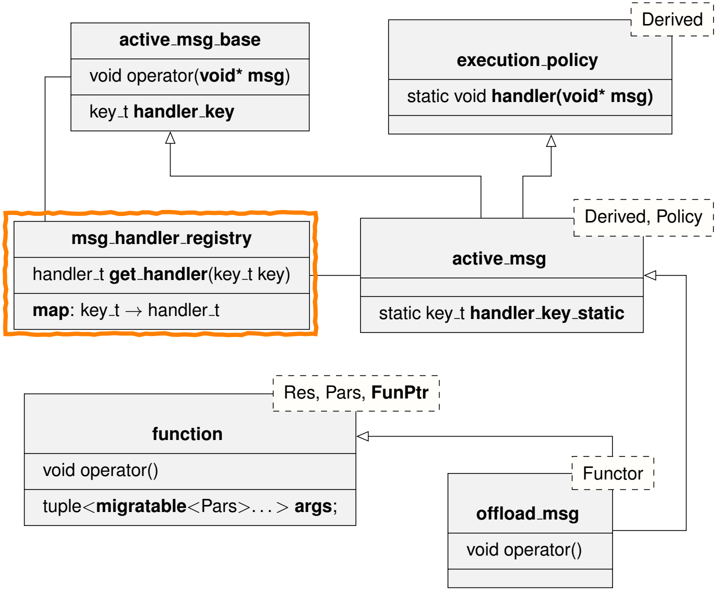
HAM Structure



`active_msg_base`

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

HAM Structure



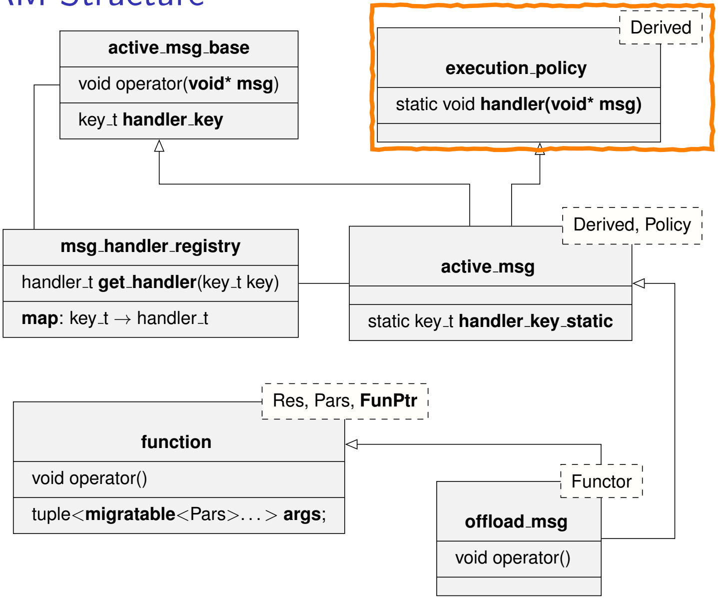
`active_msg_base`

- 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)

HAM Structure



`active_msg_base`

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

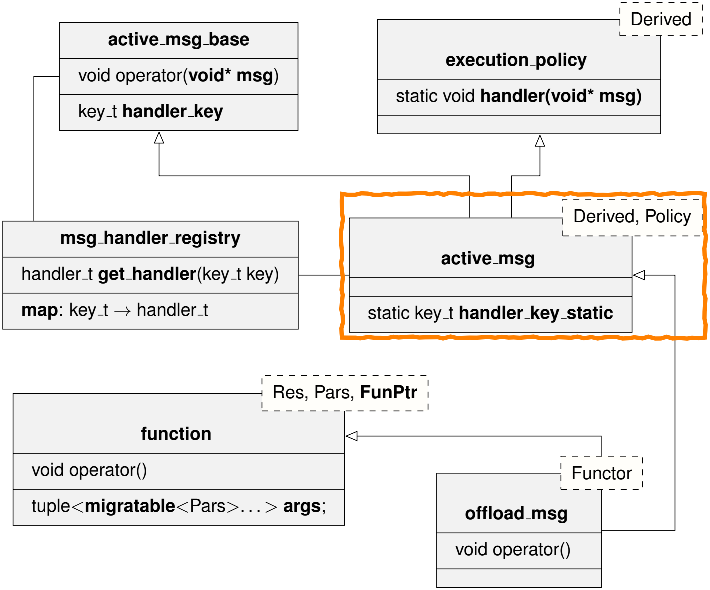
`msg_handler_registry`

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

`execution_policy`

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

HAM Structure

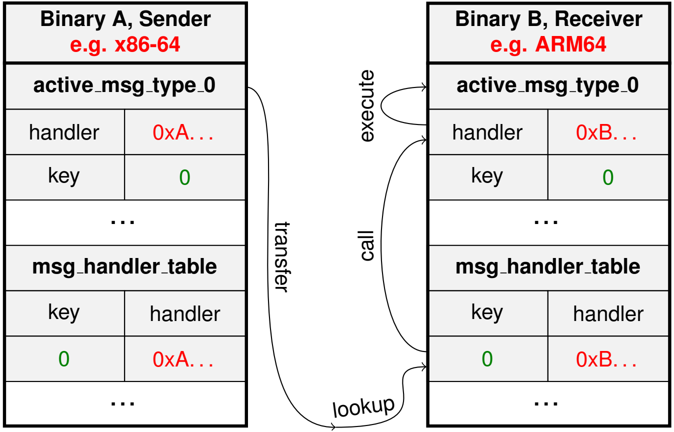


`active_msg`

- links the message type to its handler key, i.e. O(1) look-up
- static member init. provides 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_7runtime17
  terminate_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_Z7
      fun_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

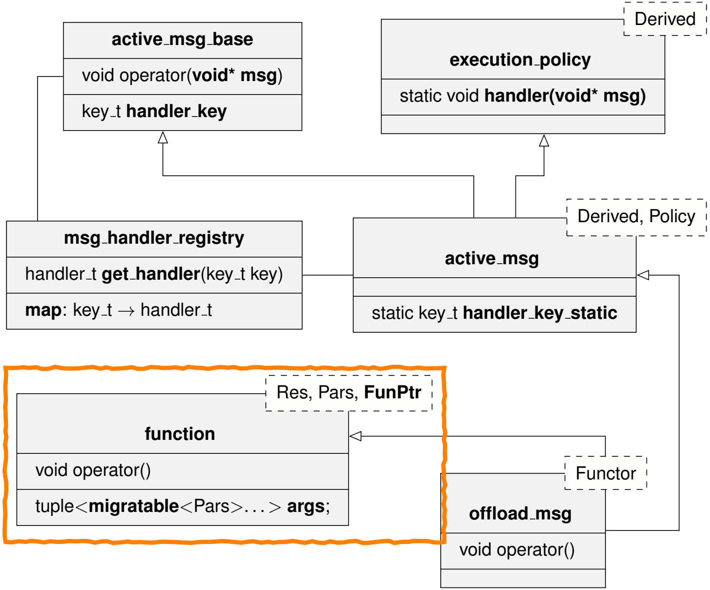
The `function` template:

```
// 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;
};
```

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`

Functions, Functors, and Lambdas

The `function` template:

```
// 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:

```
// 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:

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

Functions, Functors, and Lambdas

The `function` template:

```
// 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:

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

Hence the `f2f` (variadic macro):

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

Functions, Functors, and Lambdas

The `function` template:

```
// 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:

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

Hence the `f2f` (with C++17):

```
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:

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

// possible: unary + operator triggers
// conversion to function pointer
f2f<+[](/* Pars */){/* 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

  ⇒ can be used as template value argument

The `'+'` can be somewhat hidden:

```
// 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:

```
// some offloaded function
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_7runtime17
  terminate_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_Z7
     fun_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_7runtime17
  terminate_functorENS0_23execution_policy_directEEES7_EE
handler_address: 0x440d10
typeid_name:
  N3ham3msg10active_msgINS_7offload6detail18offload_result_msgINS_
  8functionIPFi                                 3$_08
     __invokeE E                                icyEEESC_EE
handler_address
typeid_name:
  N3ham3msg10active_msgINS_7offload6detail18offload_result_msgINS_
  8functionIPFvvEXadL_Z7
     fun_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
```

Compiler-dependent name for code generated from Lambda expression.

Summary

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

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
  - ⇒ 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

EoP

Thank you.

Feedback? Questions? Ideas?

noack@zib.de
`https://github.com/noma/ham`