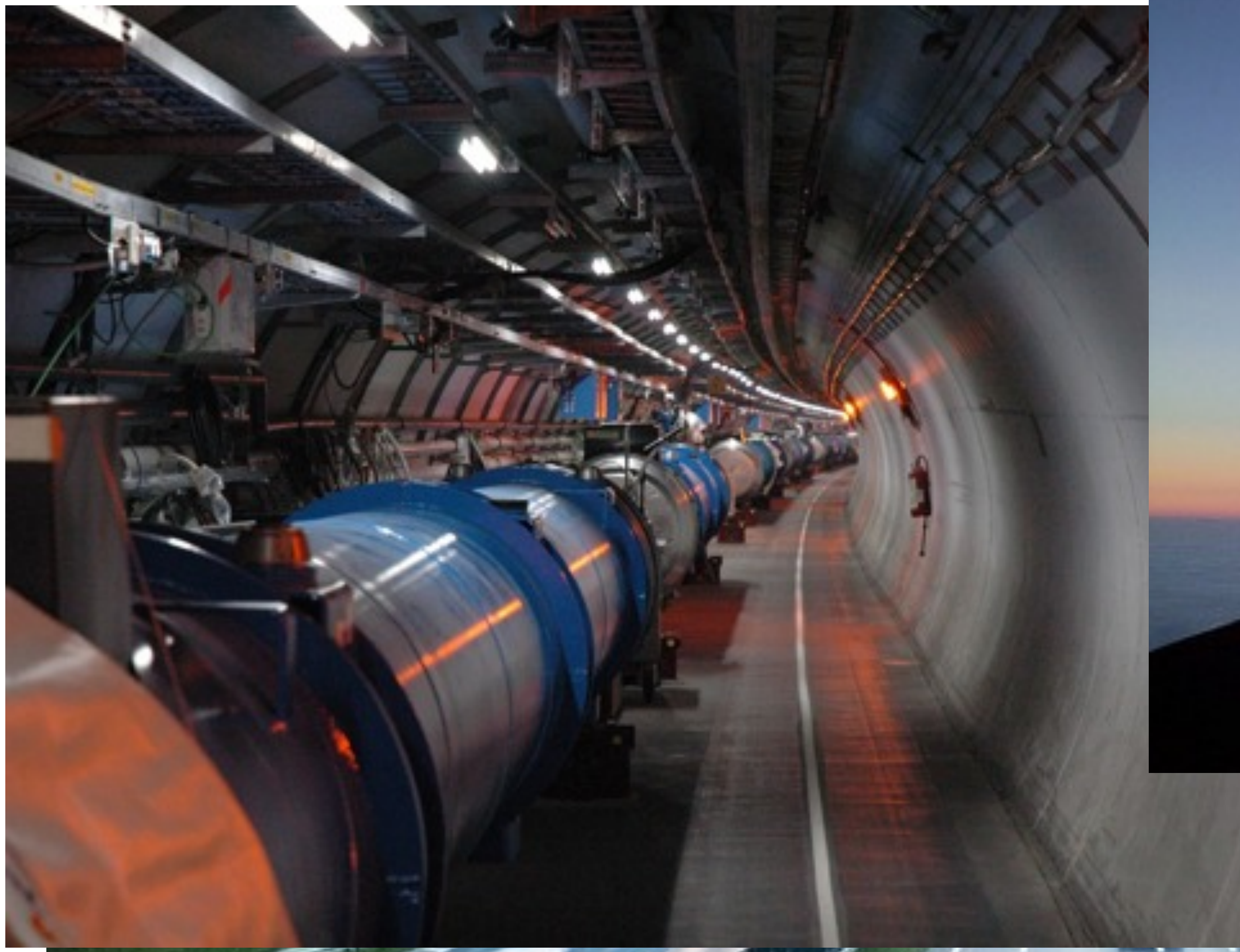
YAMI4

Messaging Solution for Distributed Systems

The Ada Connection 2011, Edinburgh, UK

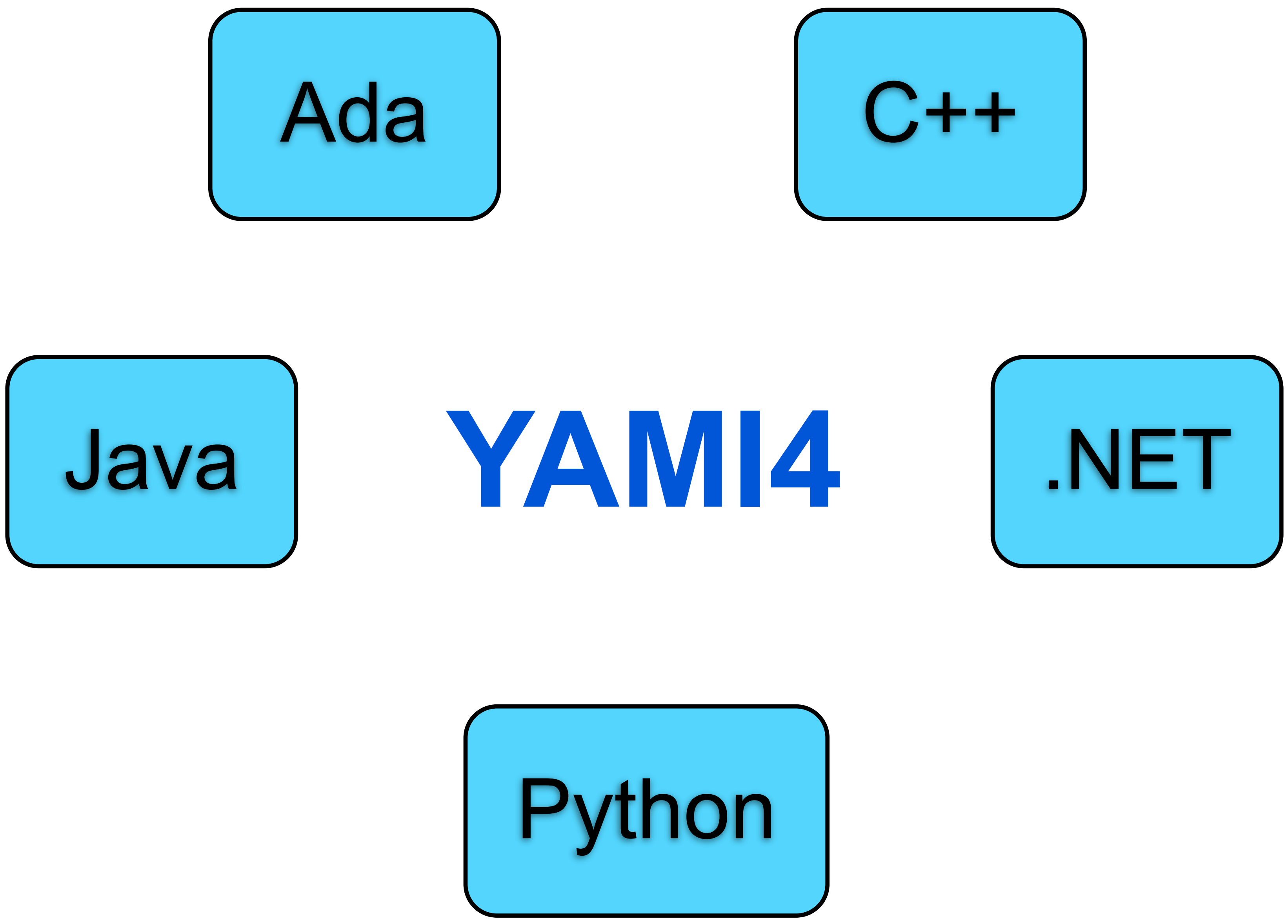
Contact: Maciej.Sobczak@inspirel.com, Inspirel, Poland

YAMI4 is a messaging solution for distributed systems and offers a range of features that make it appropriate in real-time control and monitoring installations. With asynchronous mode of communication as the most important concept, YAMI4 is scalable and ensures reasonable performance while being itself lightweight and easy to integrate with other part of the system.



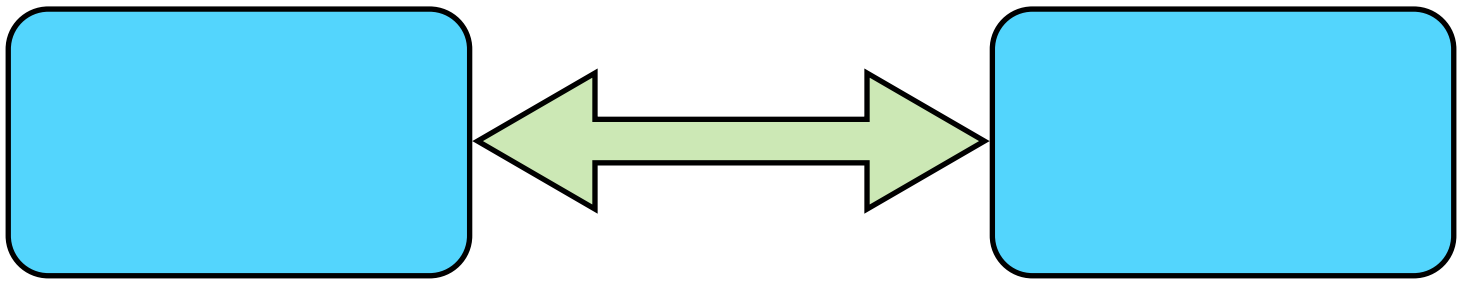
YAMI4 is used in a high-energy accelerator control system, in astronomy for remote telescope control, as well as in transport and... entertainment industry.

Library Overview



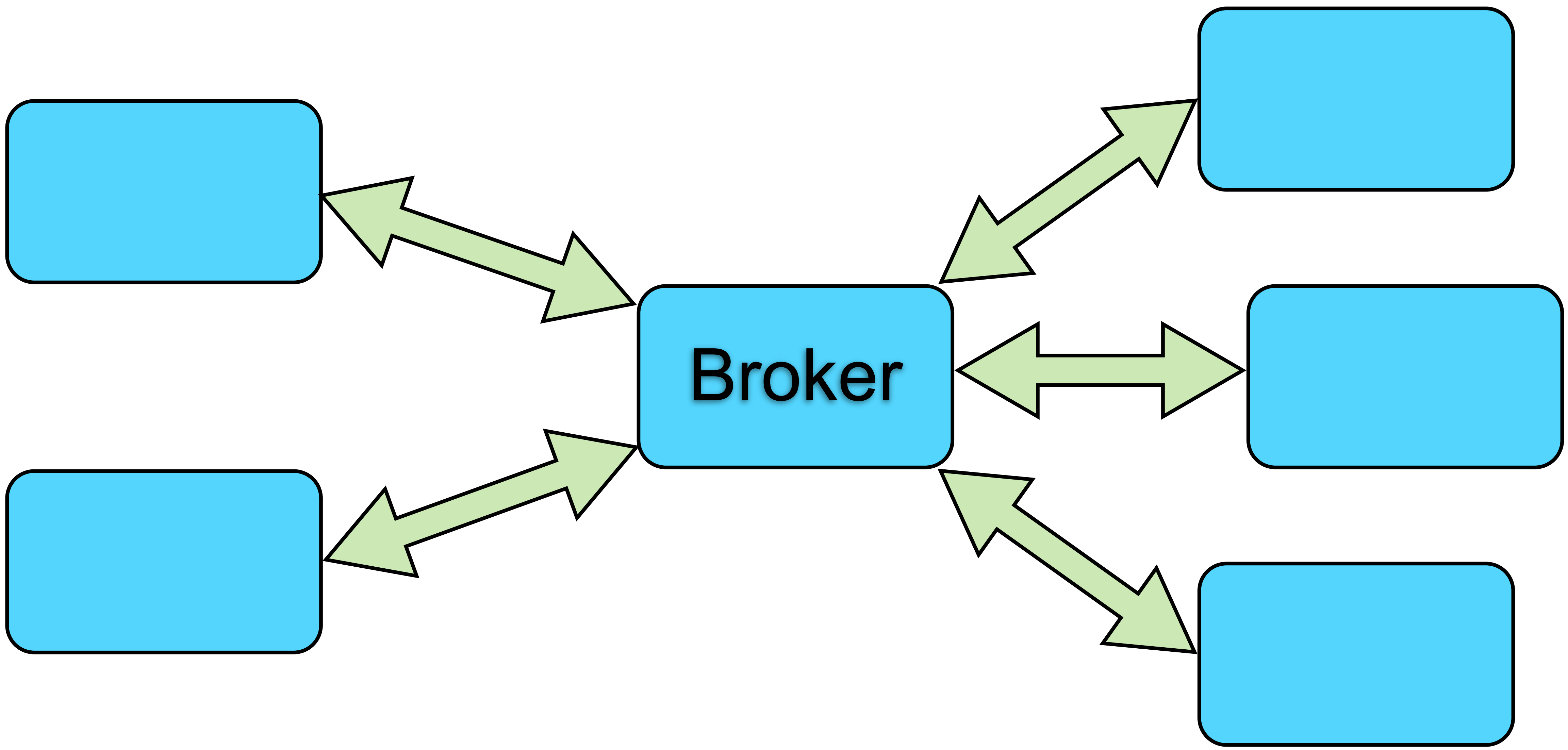
YAMI4 can be thought of as an infrastructure component that brings common services to programs implemented in different programming languages and running on different operating systems. Shared wire protocol, data model and also many of the application programming interface concepts allow to build heterogenous distributed systems.

Peer-to-peer



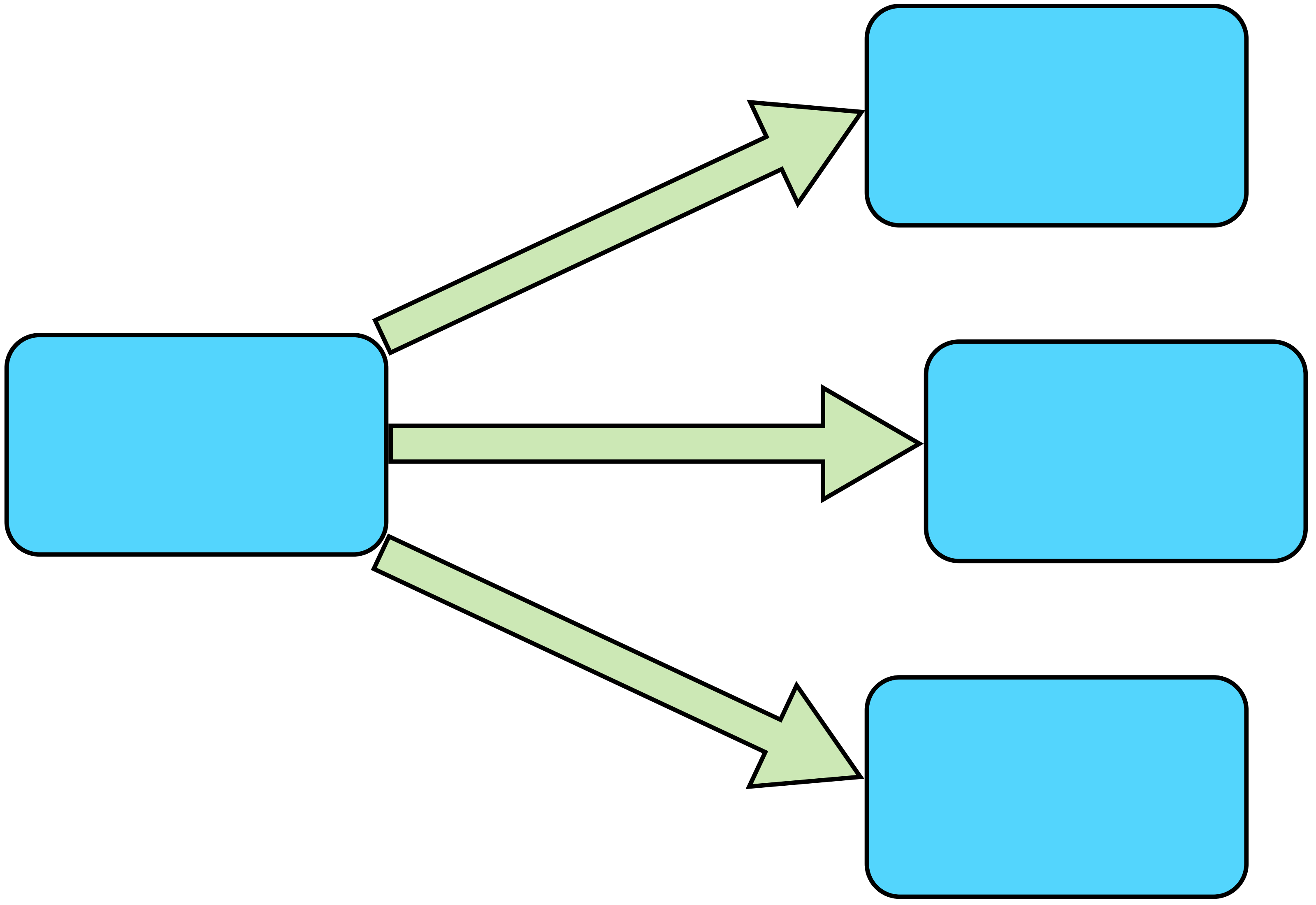
The most fundamental way of communication is a peer-to-peer channel, where nodes can exchange messages without support of any additional component like message broker. A simple client-server pattern is the most frequent example of such a system, and can be used as a building block for more elaborate architectures.

Message Broker



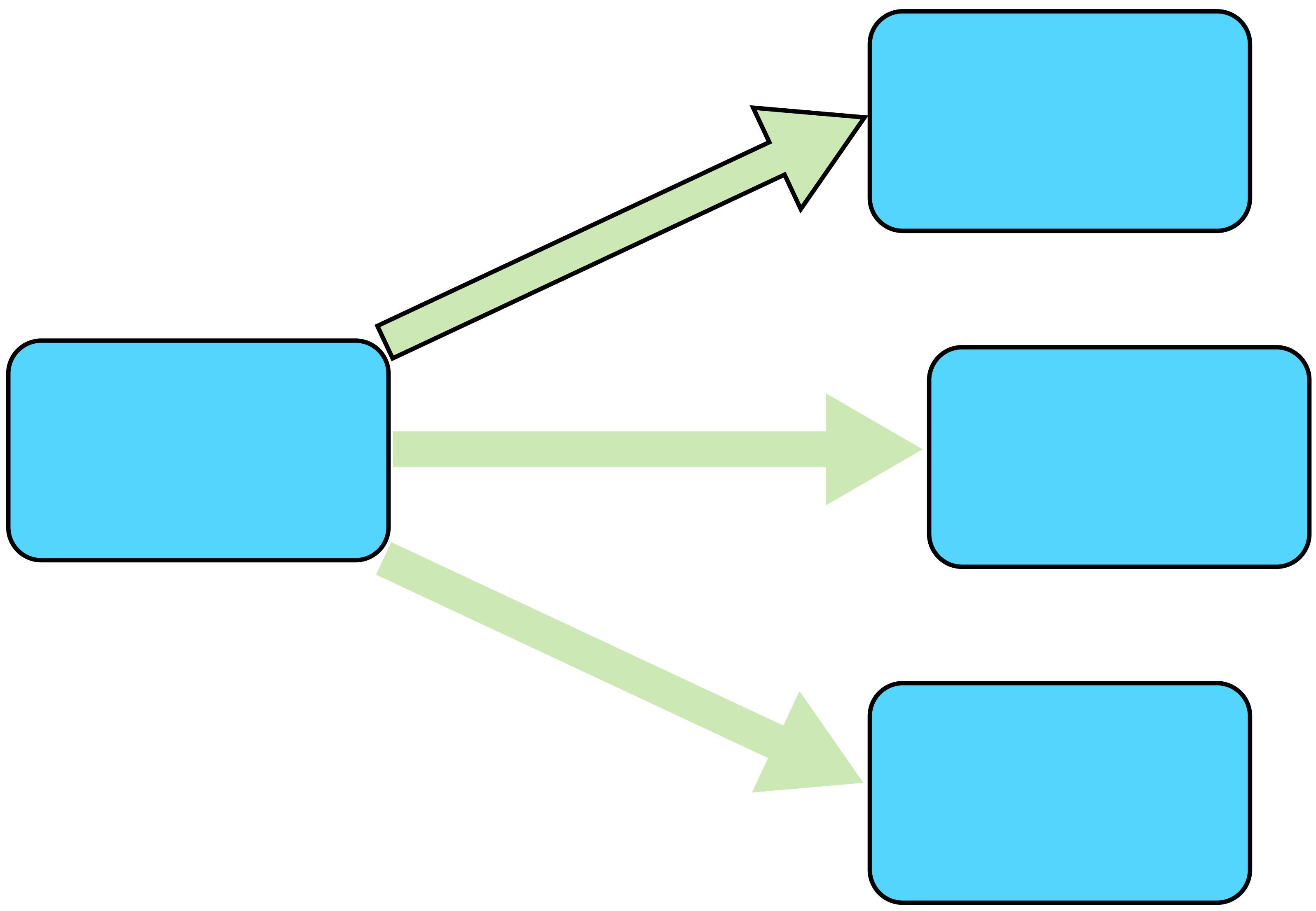
Even though not necessary for communication at the basic level, the message broker, or message service, is available as a component for application in bigger systems. It can play the role of message exchange, which delivers messages to other interested parties that communicate in the publish-subscribe manner. The broker decouples individual nodes so that they can be managed independently.

Publish-Subscribe

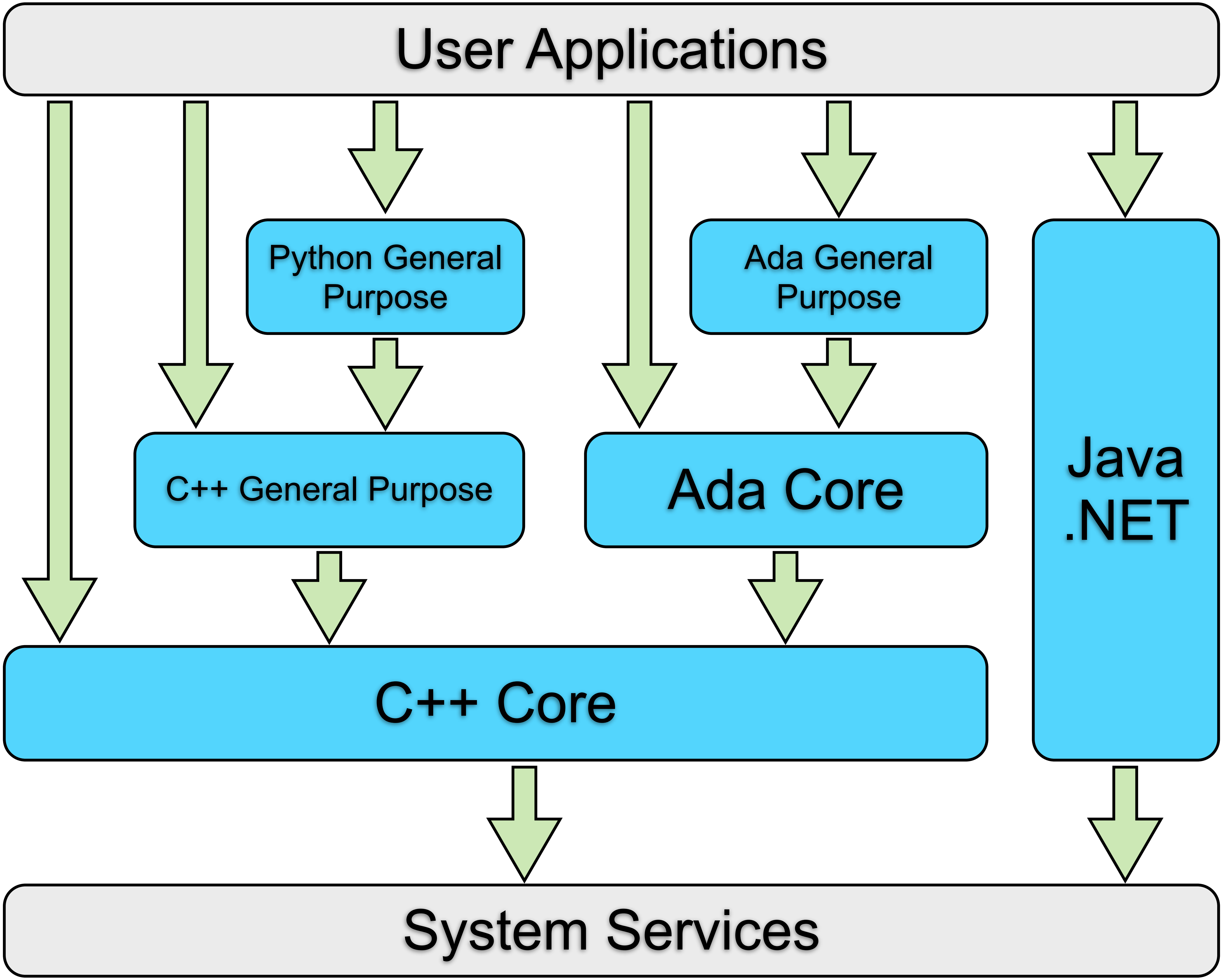


The publish-subscribe communication is also available as a high-level service in the YAMI4 library and can be used as a simple solution in those systems where a separate broker would be unnecessary administrative overhead.

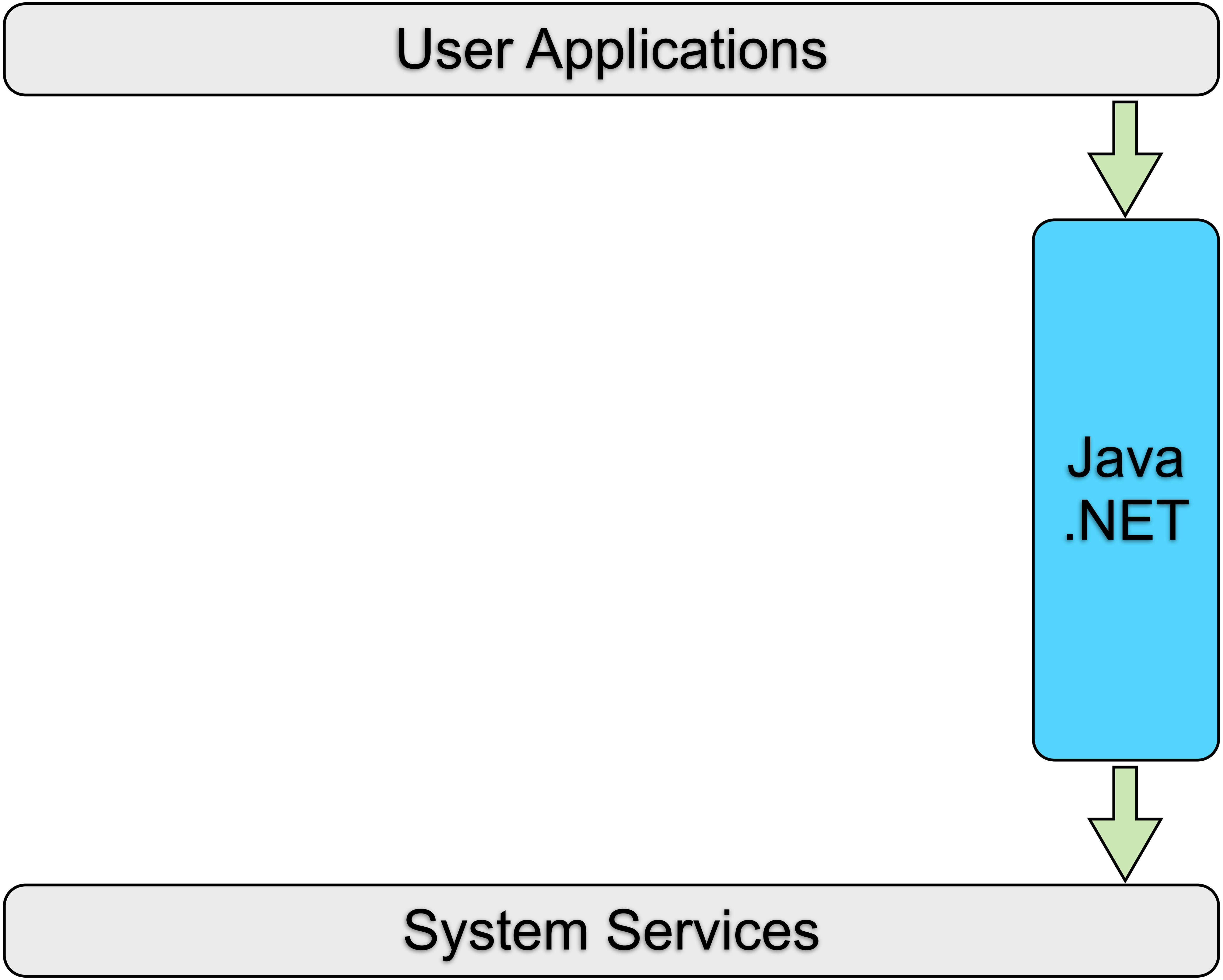
Load-Balancing



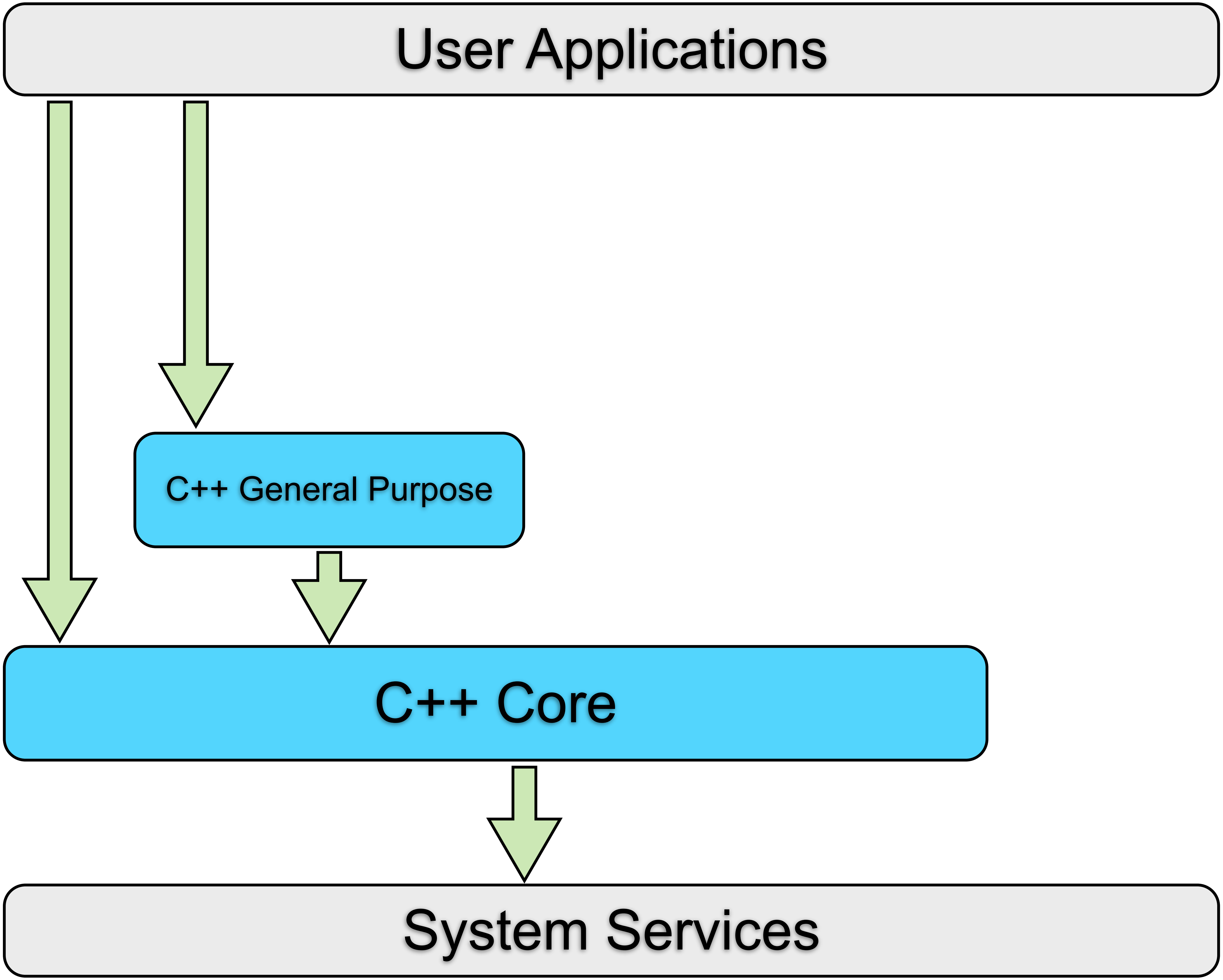
Simple load balancing and failover is supported by means of composite targets. YAMI4 randomizes the message route, which can be used for load-balancing, and automatically switches to another target if the message cannot be delivered, which can support fall-back scenario in case of failures.



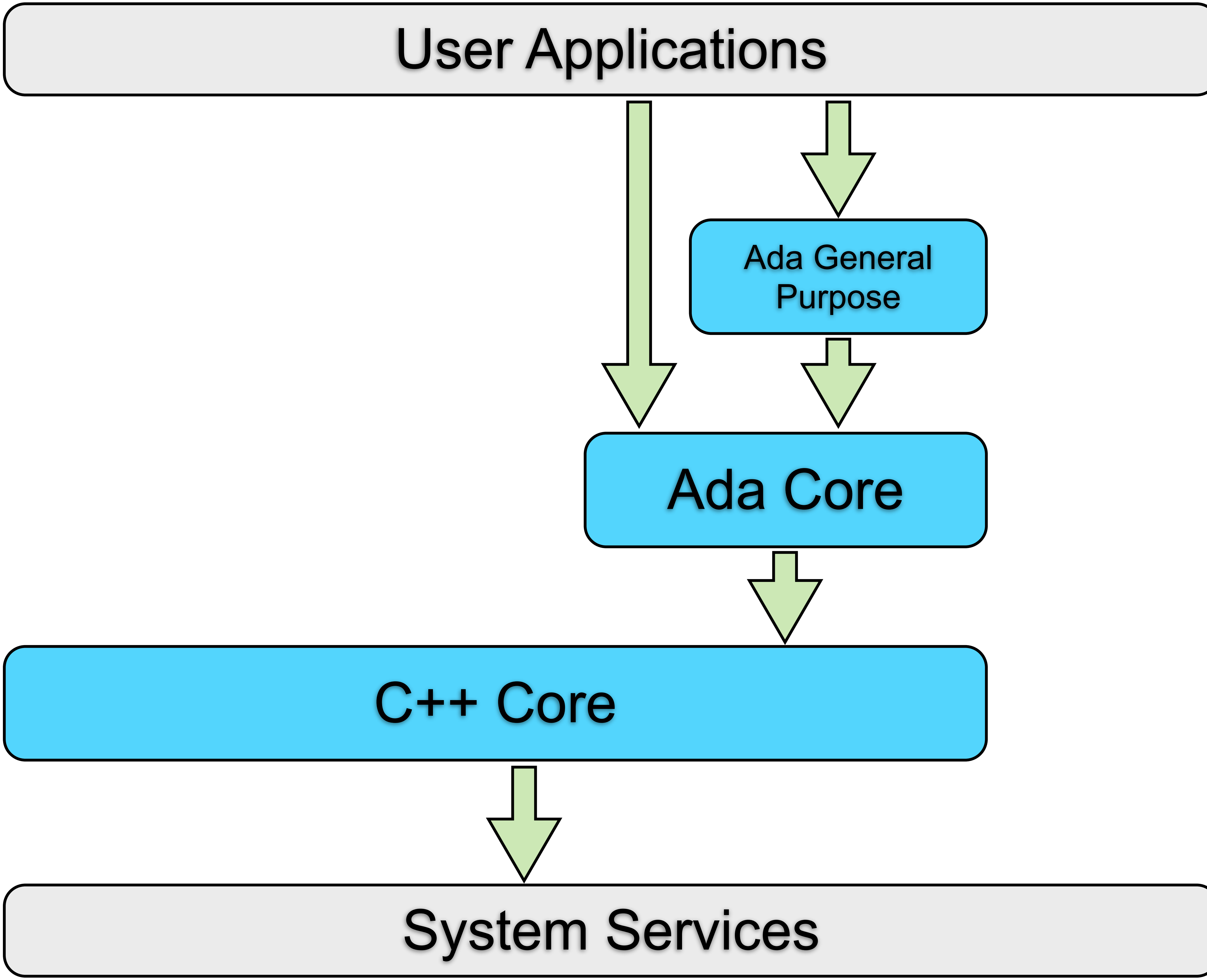
YAMI4 is a set of libraries implemented for various programming languages. Some of them are stand-alone complete implementations, like for Java or .NET, while others have layered architecture and are built on top of services provided by other components. Natively compiled languages like Ada and C++ can directly benefit from critical services provided by core libraries.



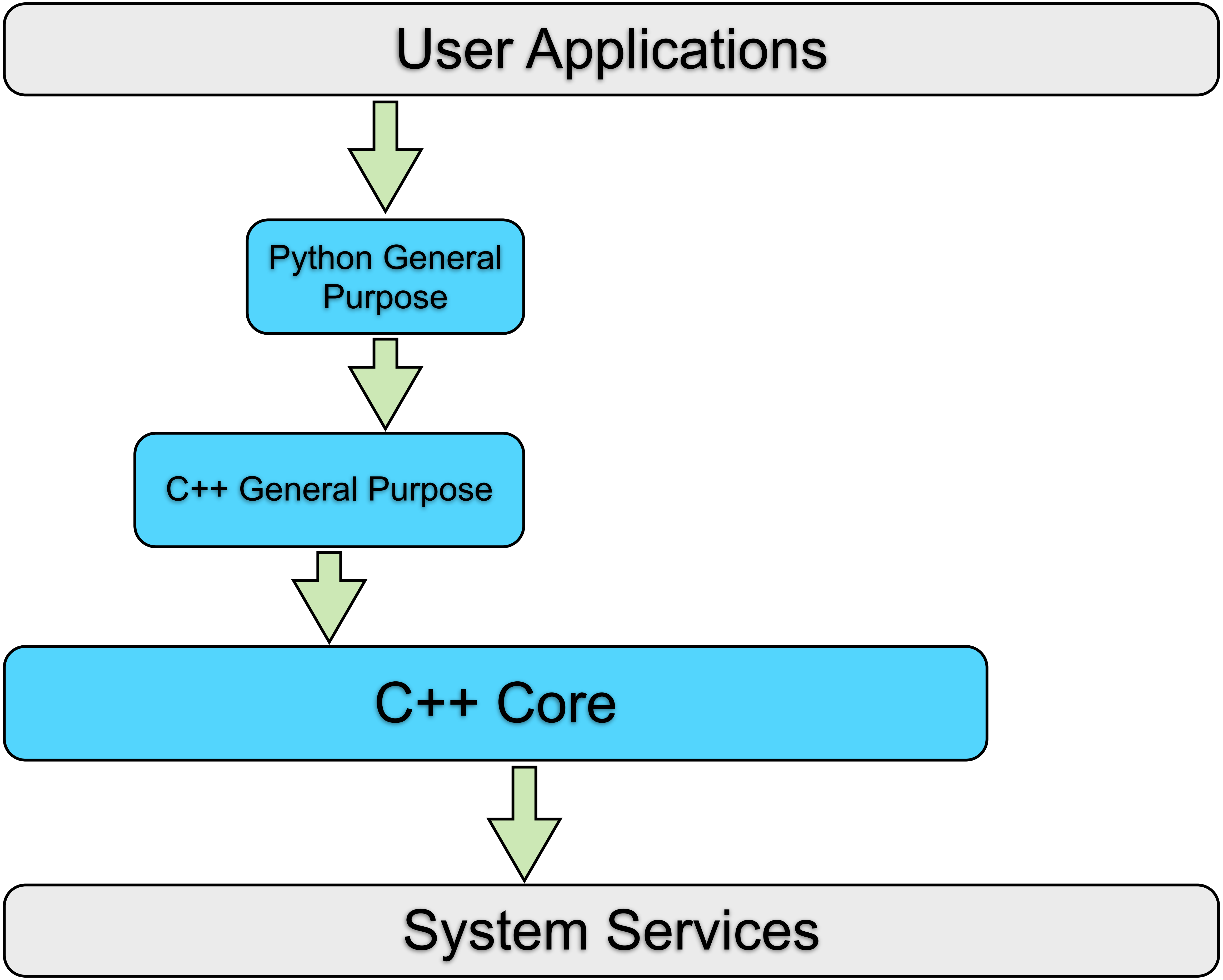
Java and .NET user programs use a single library that provides complete implementation of YAMI4 services, although with no focus on critical systems.



C++ programs can use the core component directly or benefit from high-level services provided by the general-purpose library.



Ada programs use core services via the Ada Core wrapper interface or will use general-purpose services from the higher-level library. In both cases Ada programs use and link with the core C++ component.



Python programs use a loadable module that is implemented on top of the general-purpose C++ library. This setup is not intended for critical systems, so core services are not exposed this way.

Programming Model

The YAMI4 programming model differs from the one known from various RPC solutions and is unlike the “standard” solution defined by the Distributed Systems Annex for Ada. Subsequent slides expose the issues of RPC communication and motivate an alternative solution.

RPC - what’s the advantage?

```
T := Controller.Get_Temperature;
```


Many RPC solutions build on top of syntax that is already available for local calls.

RPC - what's the advantage?

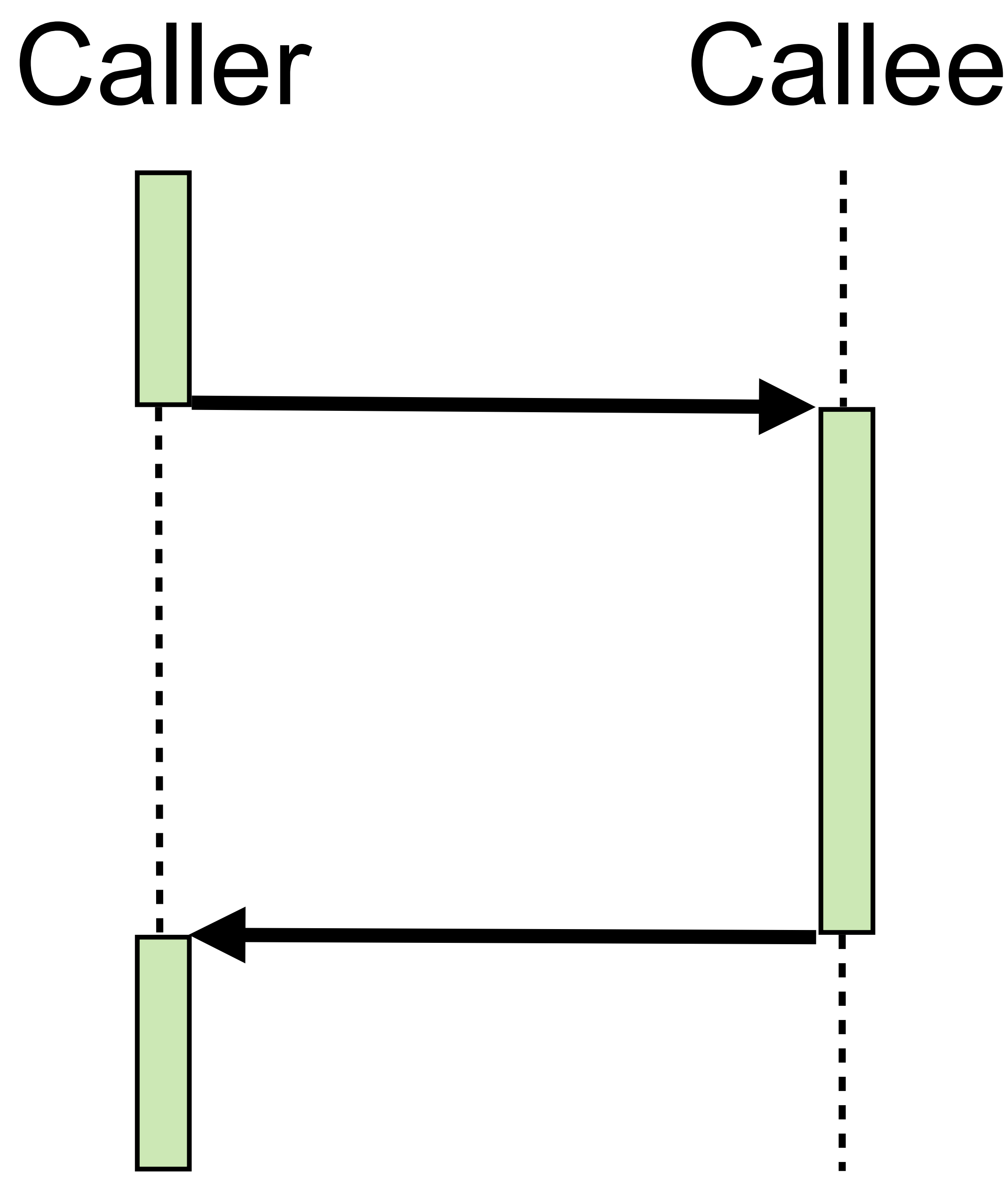
It is a natural extension of the existing concept.

The biggest advantage of this is that it naturally extends the well-known concepts and in theory makes the distribution a deployment issue instead of design issue. It is also understood by existing programming tools.

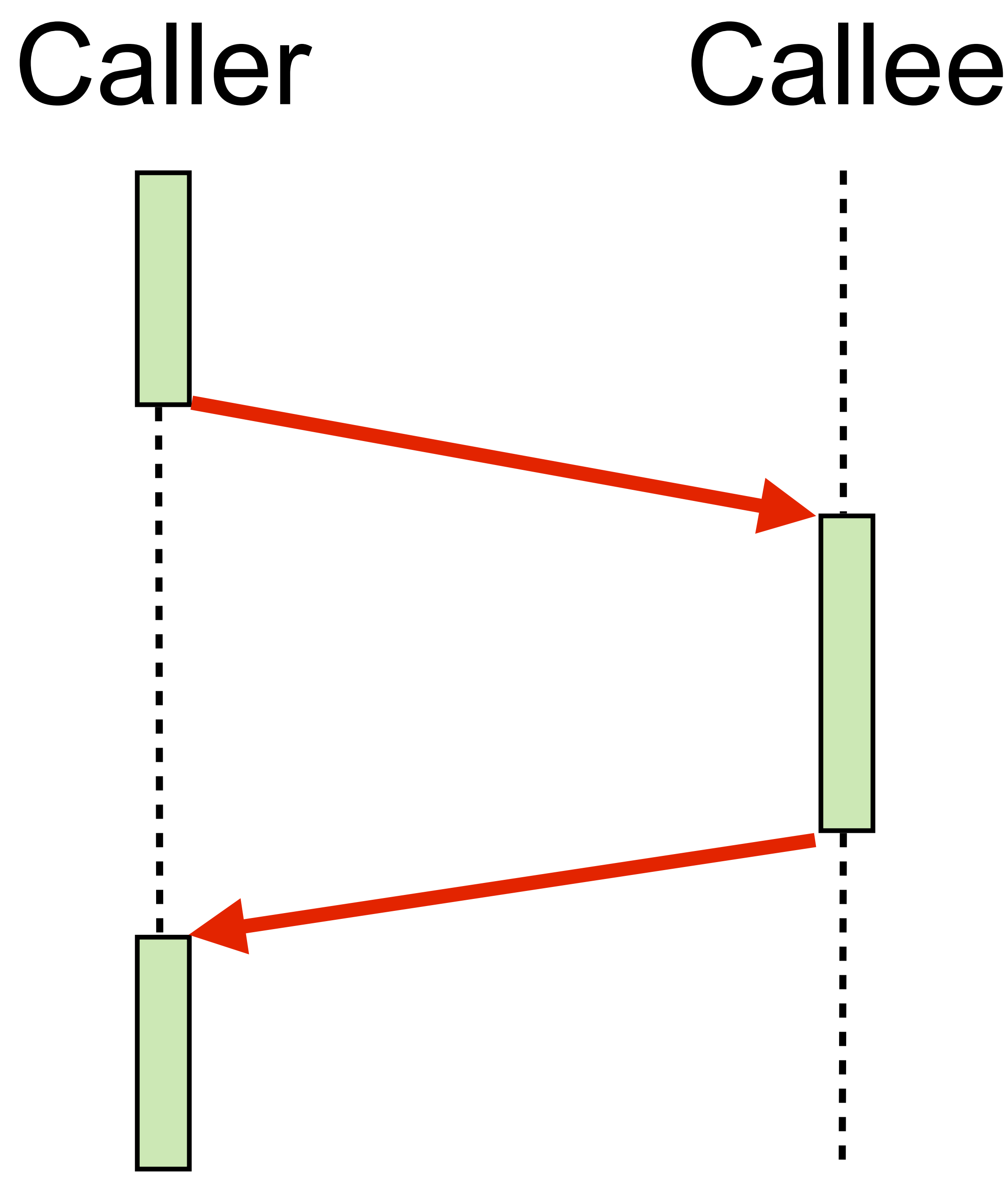
RPC - what's the problem?

```
T := Controller.Get_Temperature;
```


The problem with RPC is that the most crucial parts - the distribution and its physical aspects - are invisible and therefore implicit.



It is widely assumed that local calls are cheap - to the point of being negligible in terms of timing analysis. The control flow is supposed to be transferred between parties immediately and 100% reliably - while none of them is actually true.



In fact, in distributed systems the physicality of a call and the transfer of control flow cannot be neglected - neither in timing nor in reliability aspects.

RPC - what's the problem?

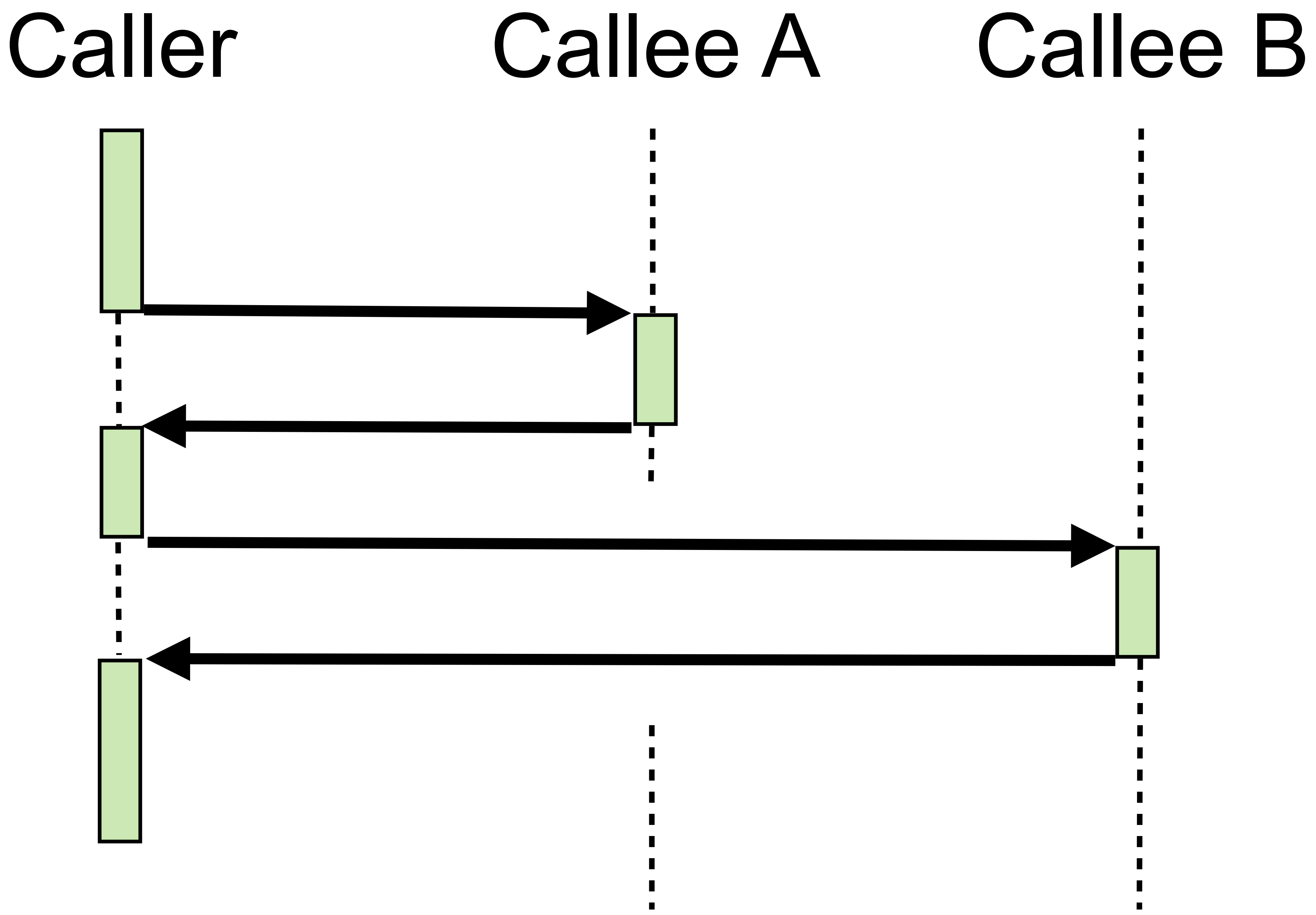
The physicality of call is implicit
- and cannot be easily managed.

The problem of RPC and related approaches is that by hiding the distribution behind the syntax that was intended for local calls they make important issues implicit and thus difficult or even impossible to manage properly.

RPC - what’s the problem?

```
T1 := Some_Controller.Get_Temperature;
T2 := Other_Controller.Get_Temperature;
```


Another issue of RPC is that orchestration of all actions depends on sequential nature of imperative syntax.



Even though distribution naturally brings concurrent capabilities (there are additional CPUs in the system), the sequential nature of RPC calls makes it difficult to benefit from this potential.

RPC - what's the problem?

It is inherently sequential
- and obstructs the concurrent nature
of the distributed system.

RPC - what's the problem?

RPC is not a proper solution for distributed systems.

The conclusion that RPC might not be a proper solution seems to be brave, but is necessary to address the requirements of accuracy, reliability and scalable performance.

Alternative to RPC

Deal with the physicality of remote call.

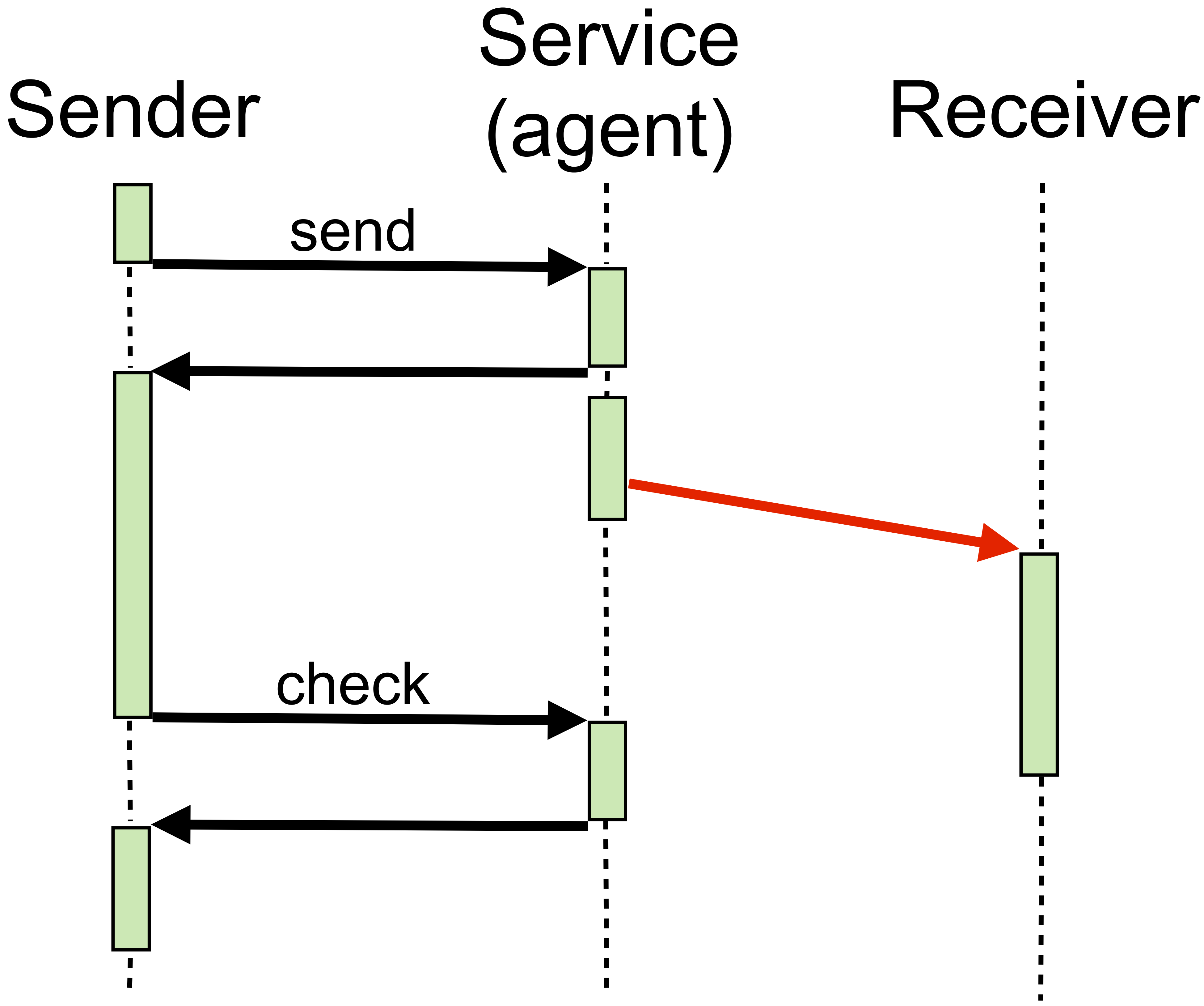
Benefit from concurrent capabilities of distributed systems.

An alternative solution for distributed systems should not hide the physicality of remote call in the sense that timing and failure of remote interaction should stay visible. In addition, a proper solution should naturally benefit from distributed concurrency without the need to mirror it in additional local tasks.

Asynchronous Messaging

Analogy: Courier Services.

YAMI4 is a messaging solution that can be best described by analogy to civil courier services, where message is a first-class entity that can be managed, monitored and even parallelized with the help of explicitly operational service.



With courier services (and with YAMI4 messaging) each of these actions is explicit and can be properly managed with regard to timing and failure. Local assumptions apply only to interactions with “nearby post office”.

Asynchronous Messaging

Agent.

Message.

Background processing.

With asynchronous messaging, as implemented by YAMI4, the user explicitly operates with entities like agent (equivalent to local post office) and message. It is also explicit that message is processed in background, concurrently to other user activity.

Asynchronous Messaging

```
T := Controller.Get_Temperature;
```

```
Msg := Agent.Send (Controller,
                   Get_Temperature);

-- ...

Msg.Wait_Until_Ready;
T := Msg.Response;
```


Obviously, the alternative to RPC cannot benefit from the local-call syntax and needs its own syntax patterns. This is the schematic comparison of what changes are needed to address the differences between RPC and asynchronous messaging.

Data Model

The data model in YAMI4 could, in principle, be built on top of existing language structures appropriately to the target language, but the existing libraries offer a uniform approach.

Why Dynamic Typing?

Because distributed systems are not statically typed.

An important property of YAMI4 is that the data model supports dynamic typing. This allows to explicitly address the fact that distributed systems cannot be statically verified for data definition consistency.

Parameters Object

Message payload is a dynamically-typed map.
Keys are strings and entries can be:

- scalars (boolean, integer, long, double)

- strings (UTF-8 strings or binary objects)

- scalar arrays

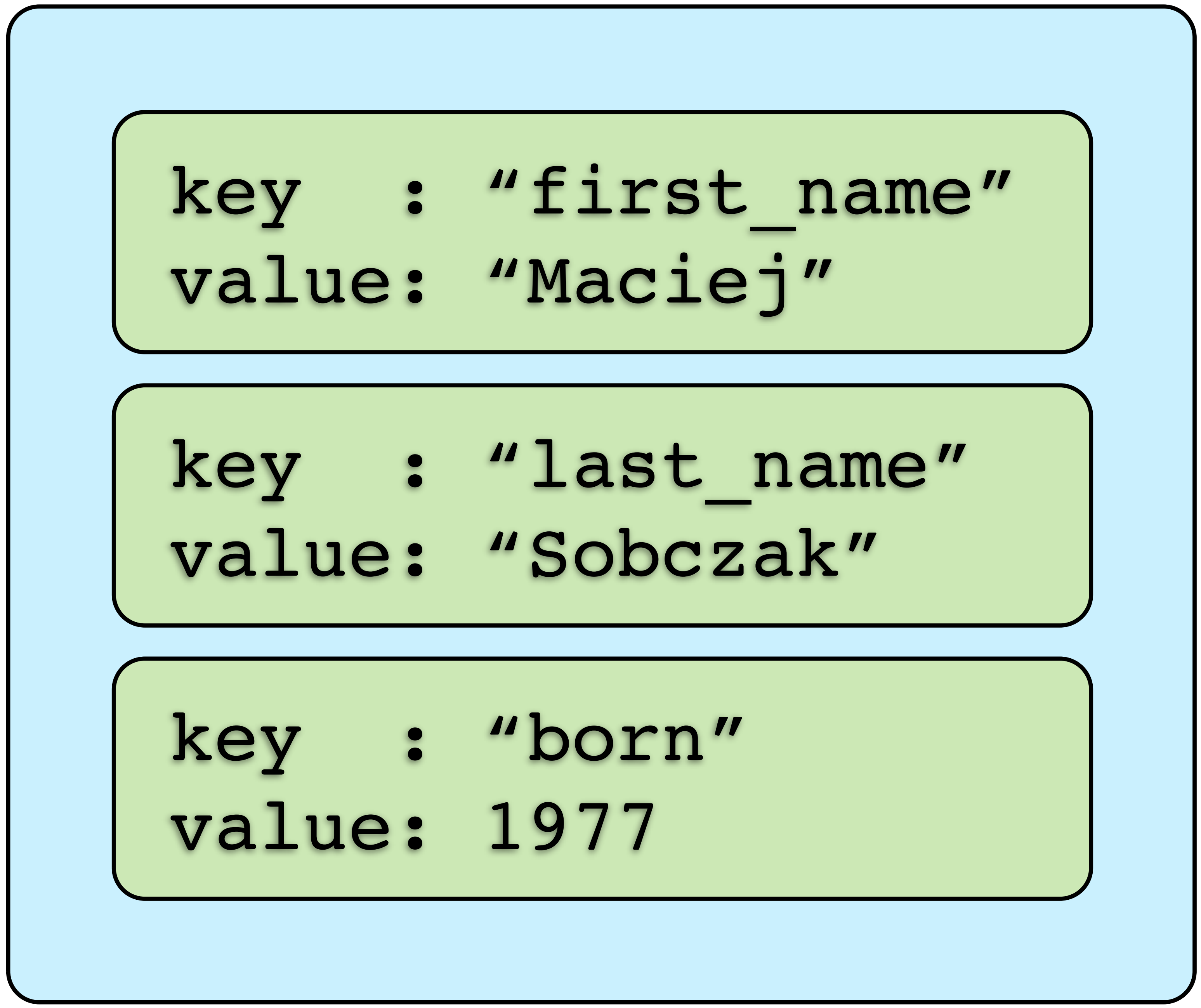
- scalar strings

- nested parameters objects

YAMI4 data model allows to build structured values from several basic types.

```
type Person is record
   First_Name : Unbounded_String;
   Last_Name : Unbounded_String;
   Born : Integer;
end record;
```


This Ada record definition can have a more or less equivalent definition in terms of YAMI4 parameters object, where individual components are described by their name and type...


key : “first_name”
value: “Maciej”
key : “last_name”
value: “Sobczak”
key : “born”
value: 1977

```
declare
   Person : Parameters_Collection :=
     Make_Parameters;
begin
   Person.Set_String ("first_name",
                      "Maciej");

   Person.Set_String ("last_name",
                      "Sobczak");

   Person.Set_Integer ("born", 1977);

   -- ...
end;
```


An example sequence builds the parameters object and fills it with named data items. Analogous code example would be very similar in each supported programming language.

```
Put_Line
  (Person.Get_String (“first_name”));

Put_Line
  (Person.Get_String (“last_name”));

Put_Line (YAMI_Integer’Image
  (Person.Get_Integer (“born”)));
```


A parameters object can be directly inspected as long as the names and types of contained items are known by the programmer. This is typically the case, but it is also possible to interact with the parameters object in a more exploratory way, if its contents and structure are not known in advance.

Parameters Object

The Parameters Object's API supports:

- **creation and destruction**
- **setting and reading**
- **entry removal**
- **iteration, search and description**
- **serialization and deserialization**

These operations are fully supported in all programming languages and provide a uniform programming interface.

Beyond Parameters Object

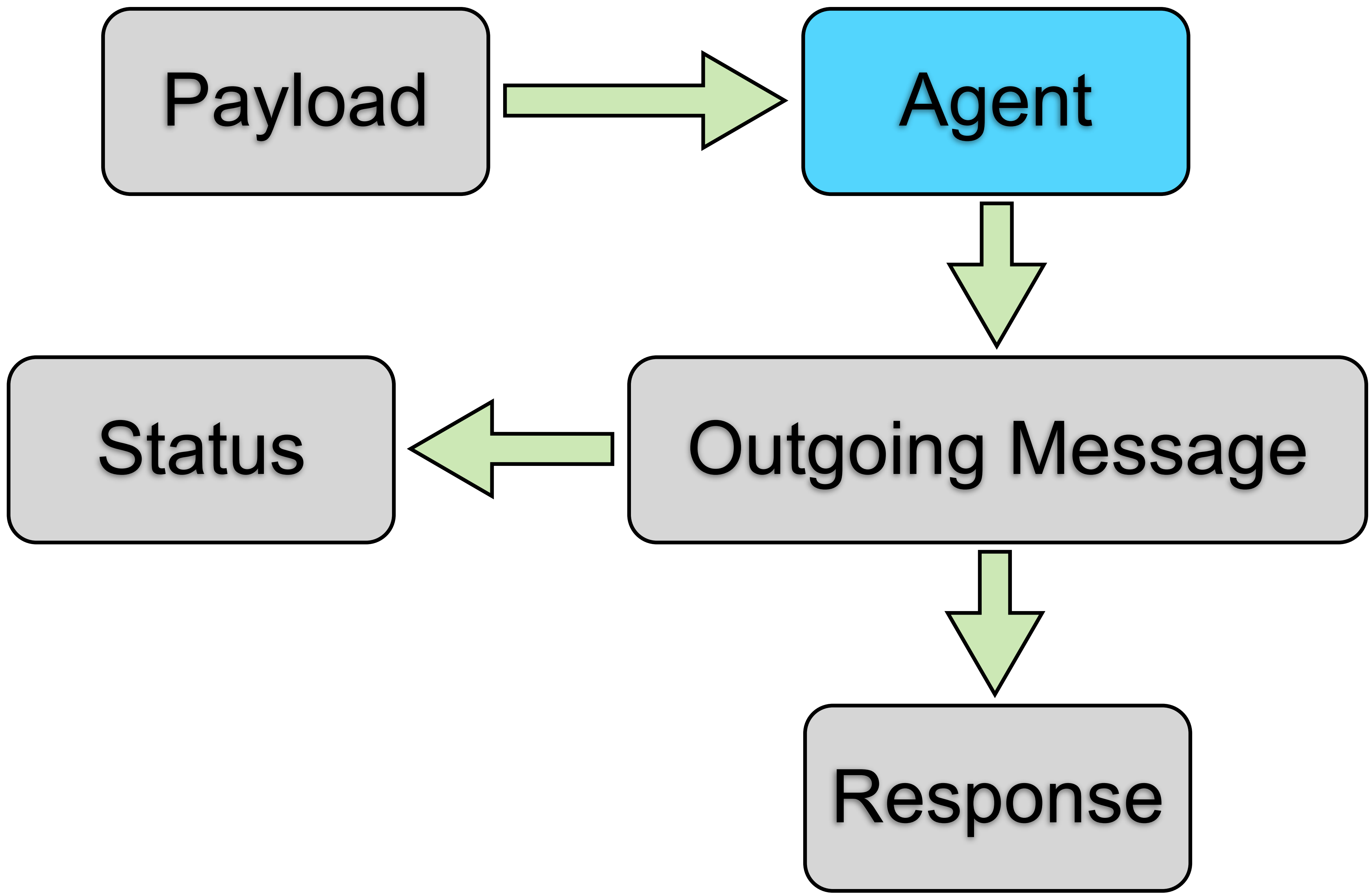
Raw binary data.

Custom serialization schemes.

In addition to the standard data model and the parameters object as its implementation, YAMI4 allows to use raw binary data for efficient transfer of opaque data as well as custom serialization schemes, which allows to integrate other models like XML, JSON, ASN.1, etc.

Sender's Perspective

Sender’s Perspective



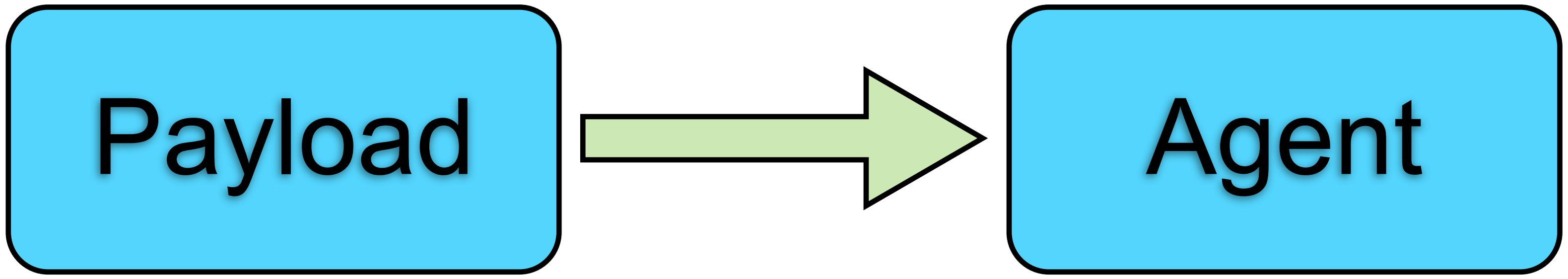
The sender in YAMI4 can use or see all of these entities, but in the simplest scenario only the agent - an equivalent to local post office - is needed. The sender interacts with the agent in order to post the new outgoing message.

Sender’s Perspective

```
declare
   My_Agent : Agent := Make_Agent;
begin
   My_Agent.Send_One_Way
     (“tcp://somewhere:12345”,
      “some_object”,
      “Hello”);
   -- ...
end;
```


In this example the sender instructs the agent to send the new message named “Hello” to the given network address, to the logical destination object (“inbox”) called “some_object”. The message does not contain any payload and its further processing is not supervised and cannot be monitored.

Sender's Perspective



The sender can also supply a payload - typically a parameters object filled with data items - to the agent. This payload will be sent as part of the message and will be available for the inspection by receiver.

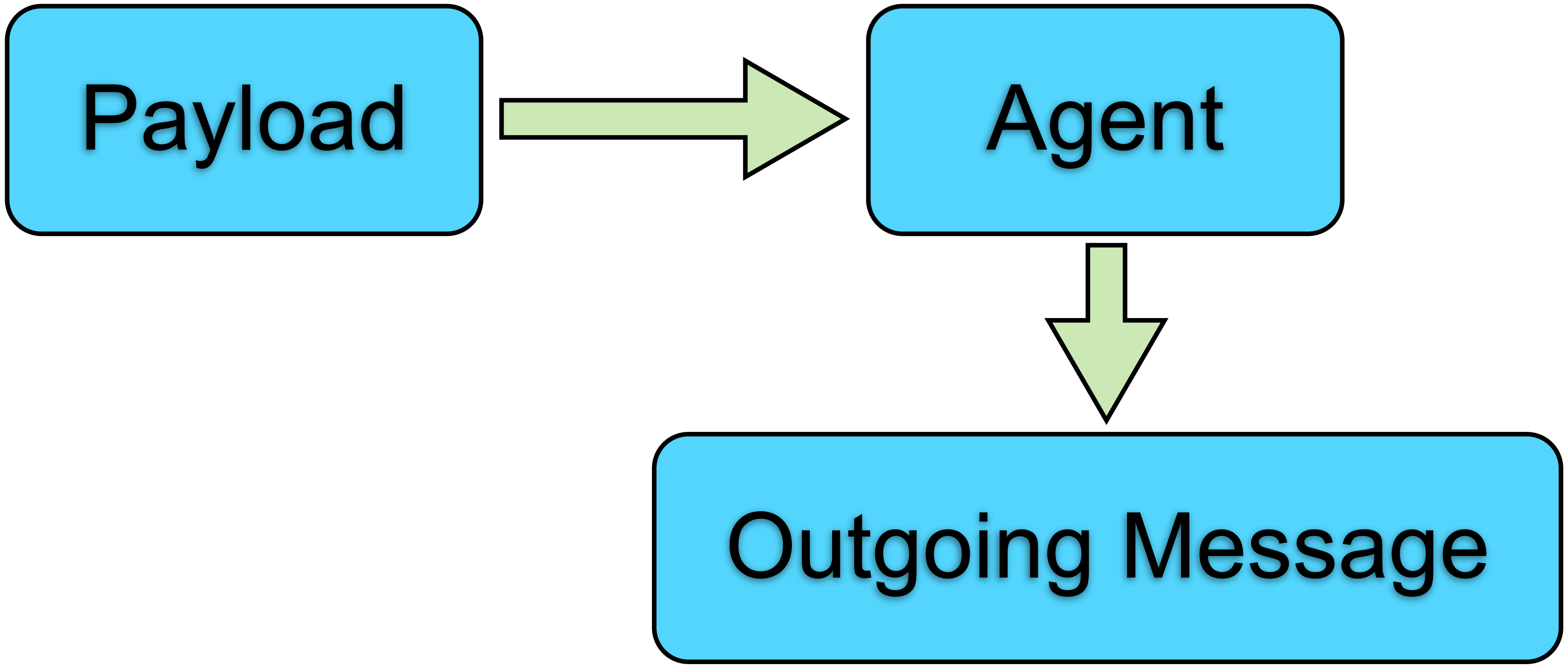
Sender’s Perspective

```
declare
   Person : Parameters_Collection :=
     Make_Parameters;
begin
   Person.Set_String (“first_name”, ...);

   My_Agent.Send_One_Way
     (“tcp://somewhere:12345”,
      “some_object”,
      “Hello”,
      Person);
   -- ...
end;
```


Message payload is supplied as an additional parameter to the post operation. After that, the parameters object is no longer needed and can be reused for further needs.

Sender's Perspective



If the message needs to be monitored or synchronized with, the agent can give back a “message receipt” - the outgoing message entity allows the sender to further observe the communication progress.

Sender’s Perspective

```
declare
   Message : aliased Outgoing_Message;
begin
   My_Agent.Send (“tcp://somewhere:12345”,
                  “some_object”,
                  “Hello”,
                  Person,
                  Message’Access);
   -- ...
   Message.Wait_For_Completion;
end;
```

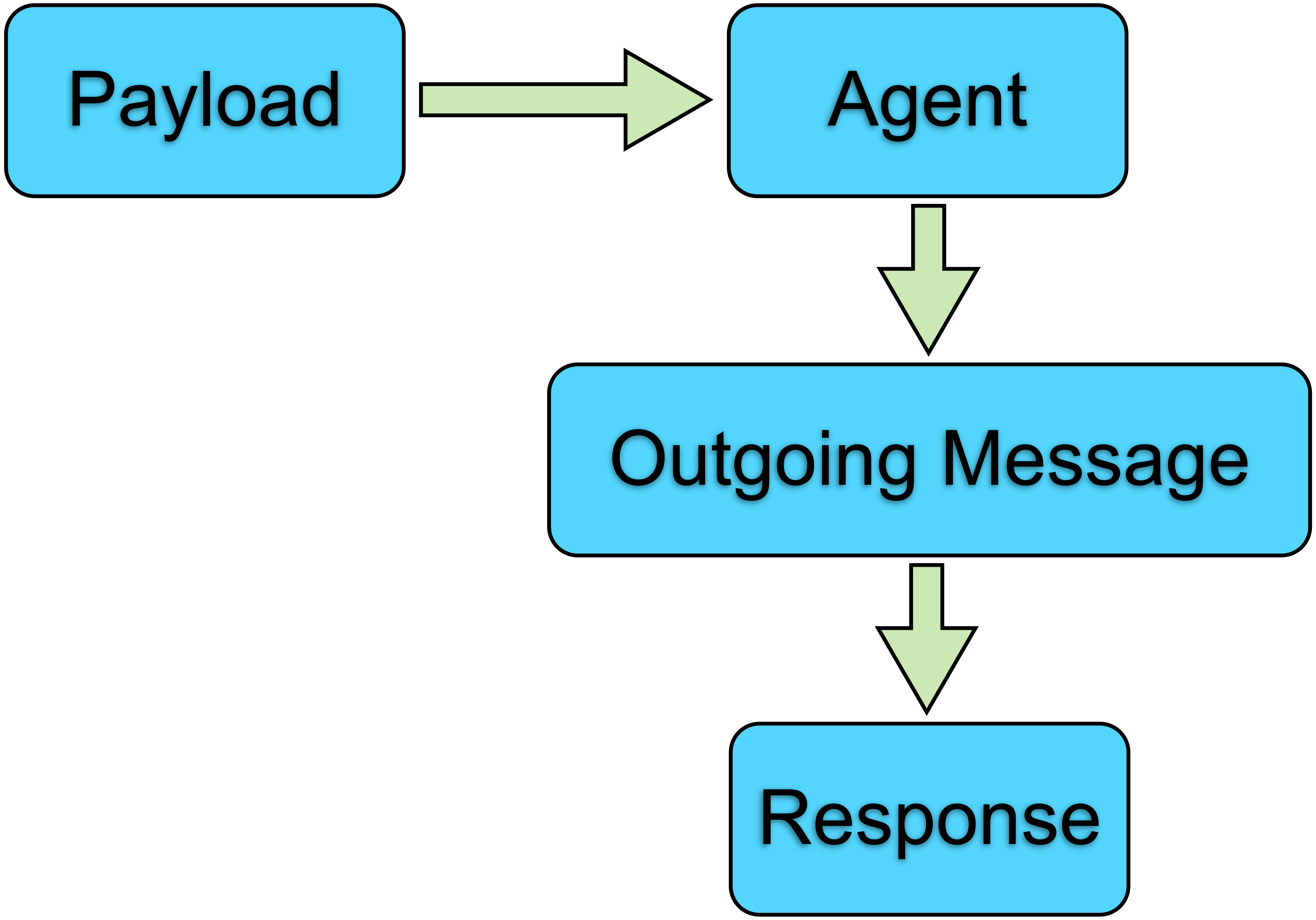

Most importantly, the sender can wait for some state change of the outgoing message - here the sender will be blocked until the message is confirmed or rejected by the receiver, which amounts to the complete client-server cycle. Waiting in Ada is a regular protected entry call and can be used in more complex patterns.

Sender’s Perspective

```
declare
   Priority : Natural := 2;
begin
   -- ...
   My_Agent.Send (“tcp://somewhere:12345”,
                  “some_object”,
                  “Hello”,
                  Person,
                  Message’Access,
                  Priority);
   -- ...
end;
```


Outgoing messages can have priorities, which are just natural numbers. Messages with higher priorities can “take over” frames of other messages that are still waiting for transmission.

Sender's Perspective



Ultimately, the sender can inspect the response that was sent back from the receiver - the response is a regular parameters object.

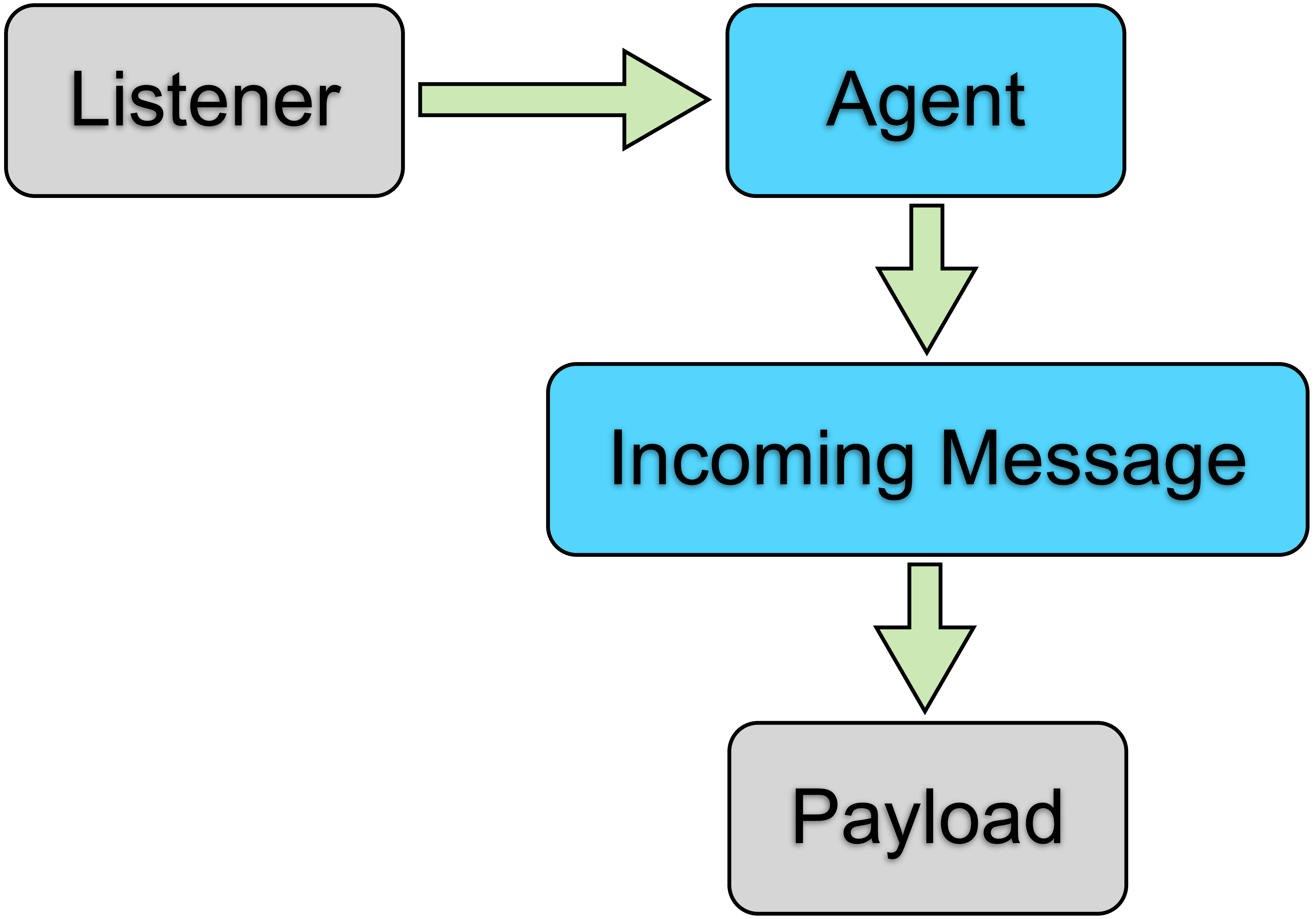
Sender's Perspective

```
declare
   procedure Process_Reply (
      Response : in out Parameters_Collection) is
   begin
      Put_Line (Response.Get_String ("address"));
      -- ...
   end Process_Reply;
begin
   -- ...
   Message.Process_Reply_Content
      (Process_Reply'Access);
end;
```


Message response can be processed by a callback. Here a local procedure is used to inspect the response payload.

Receiver’s Perspective

Receiver's Perspective



The message receiver - for example a server in a client-server system - always interacts with at least two entities: the agent, which accepts remote connections and manages the traffic, and the incoming message object for each received message.

Receiver’s Perspective

Listener allows the agent to accept new connections.

A typical server-side agent has at least one listener.

Listeners are created by agents on user demand and are responsible for accepting new incoming connections. They are typically needed by server-like programs, but are not necessary for handling incoming traffic on already created connections.

Receiver's Perspective

```
"tcp://here:12345"
"tcp://here:*"
"tcp://*:12345"
"tcp://*:*"

"udp://..."

"unix://localpath"
```


Listener targets can be defined in many ways and support TCP/IP, UDP and local Unix sockets. Wildcards support system-assigned port numbers and allow to work with multiple network interface cards.

Receiver’s Perspective

```
declare
   My_Agent : Agent := Make_Agent;
   Resolved :
     String (1 .. Max_Target_Length);
   Resolved_Last : Natural;
begin
   My_Agent.Add_Listener
     (“tcp://here:12345”,
      Resolved, Resolved_Last);
   -- ...
end;
```


This code example shows the typical initialization steps of the server program - if the given listener endpoint contains wildcards, it is resolved so that complete target can be passed around to other programs or stored in the name service.

Receiver’s Perspective

Message handler allows the user code to receive incoming messages.

In Ada the handler is an implementation of a callback interface.

The actual message delivery happens via the callback mechanism. The user provides his own implementation of the handler interface and registers it for the given logical object name or “inbox” - incoming messages will be automatically delivered by a dispatching call to the message handler.

Receiver's Perspective

```
type Message_Handler is limited interface;

procedure Call
   (H : in out Message_Handler;
    Message : in out Incoming_Message'Class)
   is abstract;
```


Incoming message handler is a simple interface with just one primitive operation...

Receiver's Perspective

```
declare
   type My_Handler_Type is
     new Message_Handler with null record;

   overriding procedure Call
     (H : in out My_Handler_Type,
      Msg : in out Incoming_Message'Class) is
   begin
     -- ...
   end Call;

   -- ...
begin
    -- ...
end;
```


... which allows the user to implement arbitrary message handling logic.

Receiver's Perspective

Message handler has to be registered for the given "object", or "inbox" name.

It is possible to define just one message handler in the program and use it for all incoming messages, but in more elaborate scenarios different handlers can be registered for distinct logical destination objects. In a typical system handlers are instantiated and registered once at initialization time although more dynamic associations are also possible.

Receiver’s Perspective

```
declare
   type My_Handler_Type is
     new Message_Handler with null record;
   -- ...

   My_Handler : aliased My_Handler_Type;
begin
   -- ...

   My_Agent.Register_Object
     (“my_object”, My_Handler’Access);
   -- ...
end;
```


This is a simplified code structure - in fact, message handlers are typically defined and instantiated at library level, where they live till the end of the program. This allows them to be aliased and accessed by the agent(s) with no risk of dangling references.

Receiver’s Perspective

The incoming message’s API supports:

- **inspection: object name, message name, source location**

- **reading the payload**

- **rejection**

- **reply with or without response**

Receiver’s Perspective

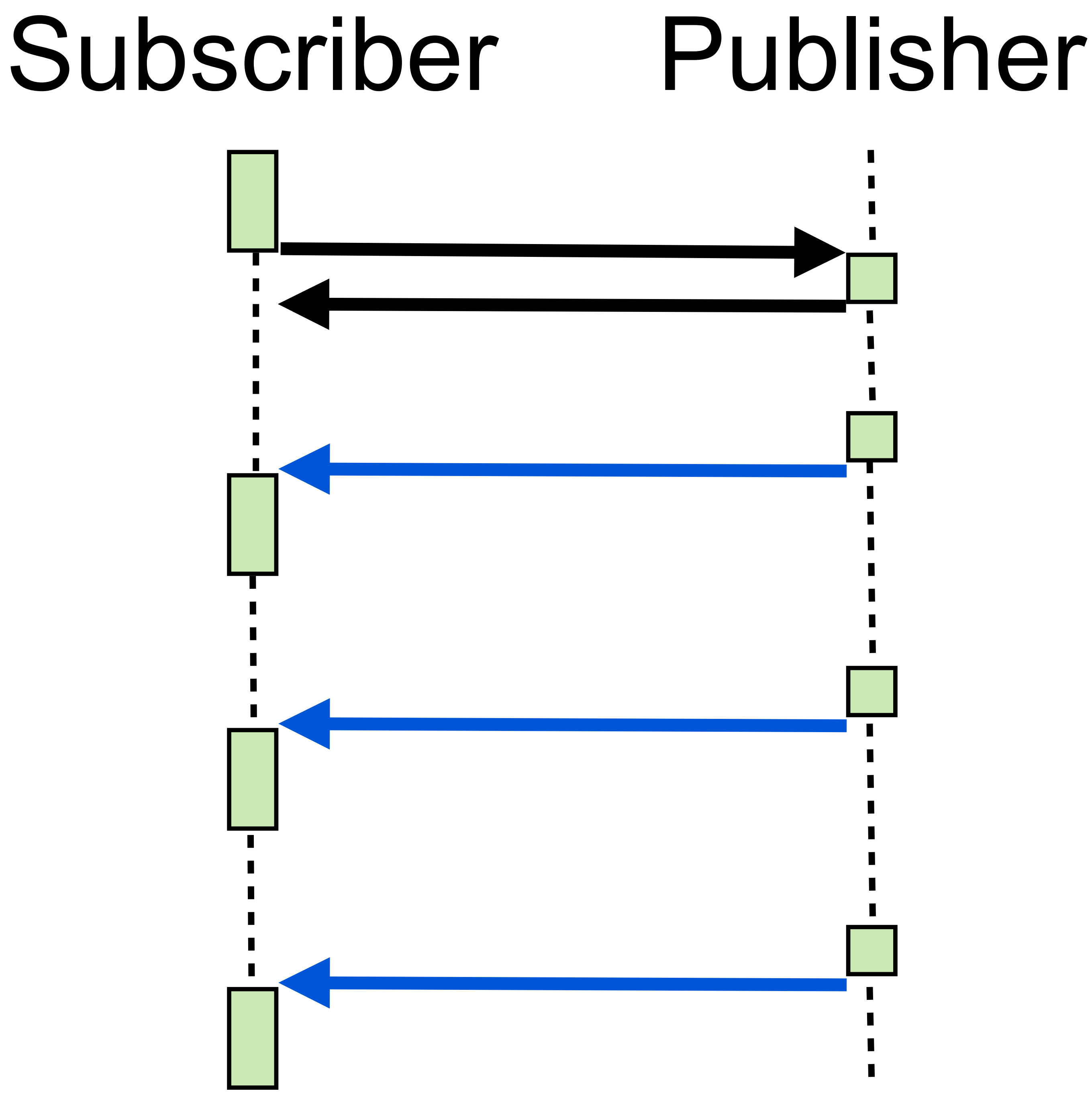
```
declare
   -- ...
   overriding procedure Call
     (H : in out My_Handler_Type,
      Msg : in out Incoming_Message'Class) is
     Response : Parameters_Collection :=
       Make_Parameters;
   begin
     -- ...
     Msg.Reply (Response);
   end Call;
   -- ...
begin
   -- ...
end;
```


Receiver will typically process the incoming message and reply to it within the message handler. All this activity happens in the context of agent dispatcher task(s).

Big Picture Services

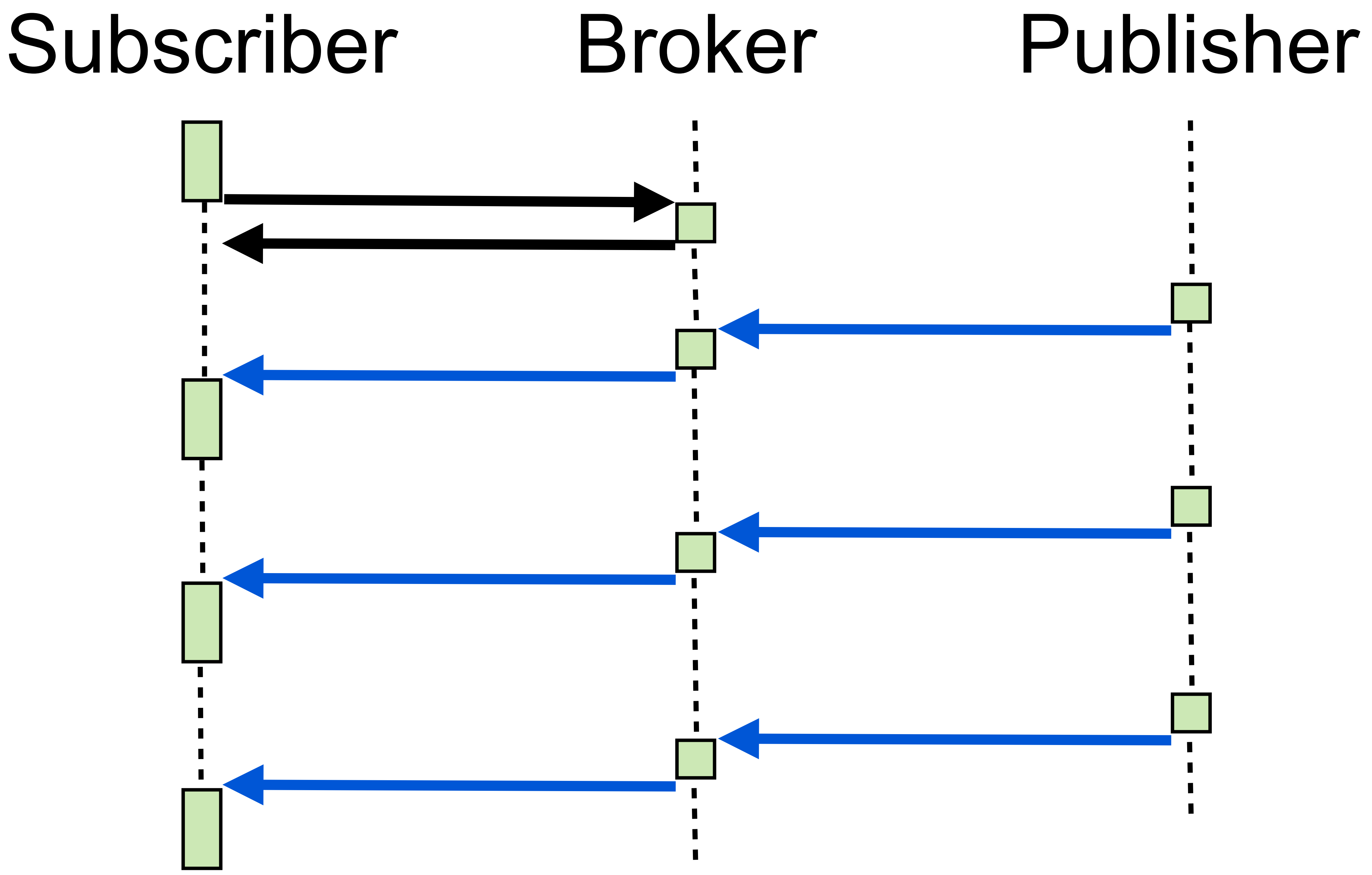
Several standard services are provided as part of the YAMI4 suite to act as building blocks for bigger distributed systems.

Direct Publish-Subscribe



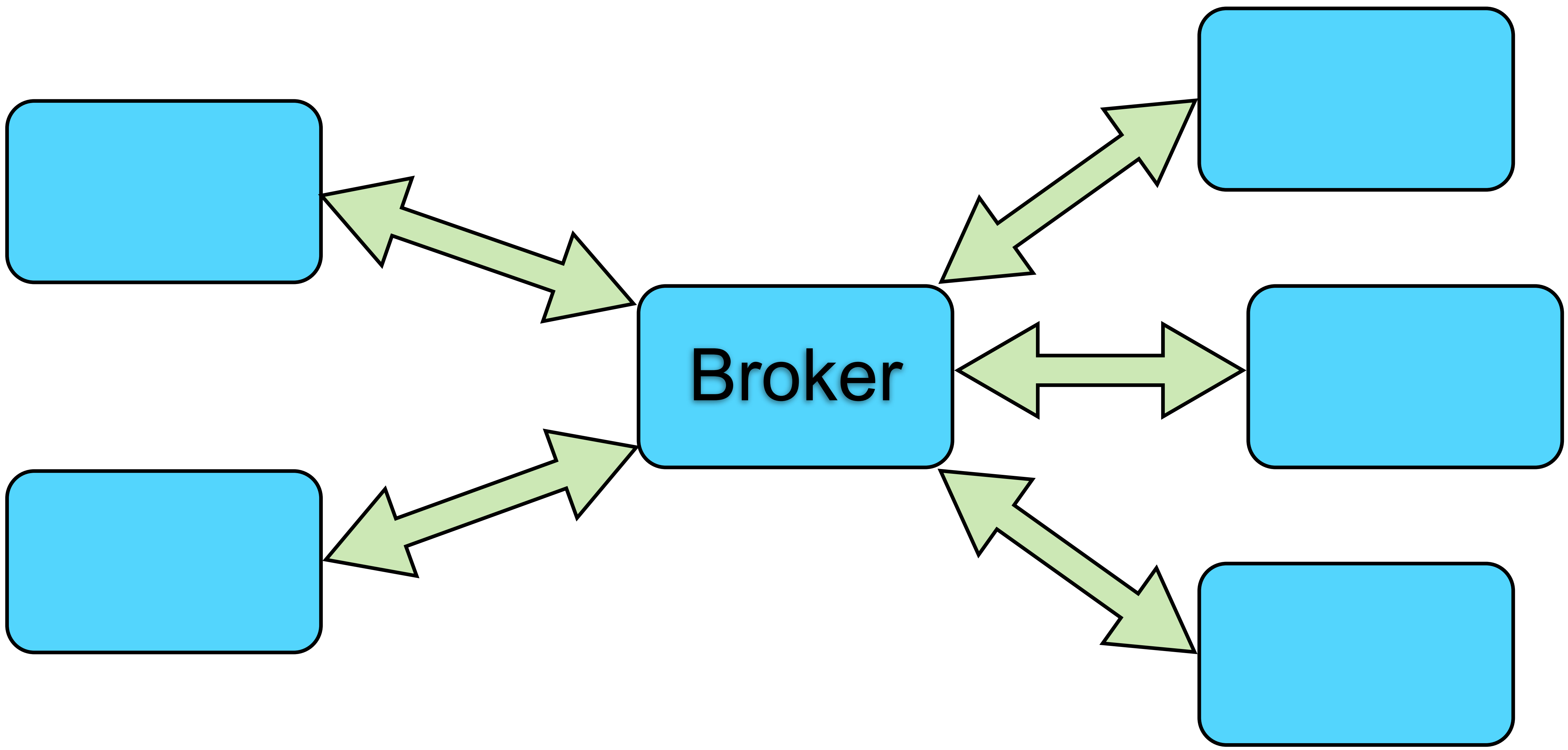
The publish-subscribe service is supported for both direct connections and for use with external broker. Direct pub-sub is appropriate for state monitoring with small number of subscribers.

Pub-Sub with Broker



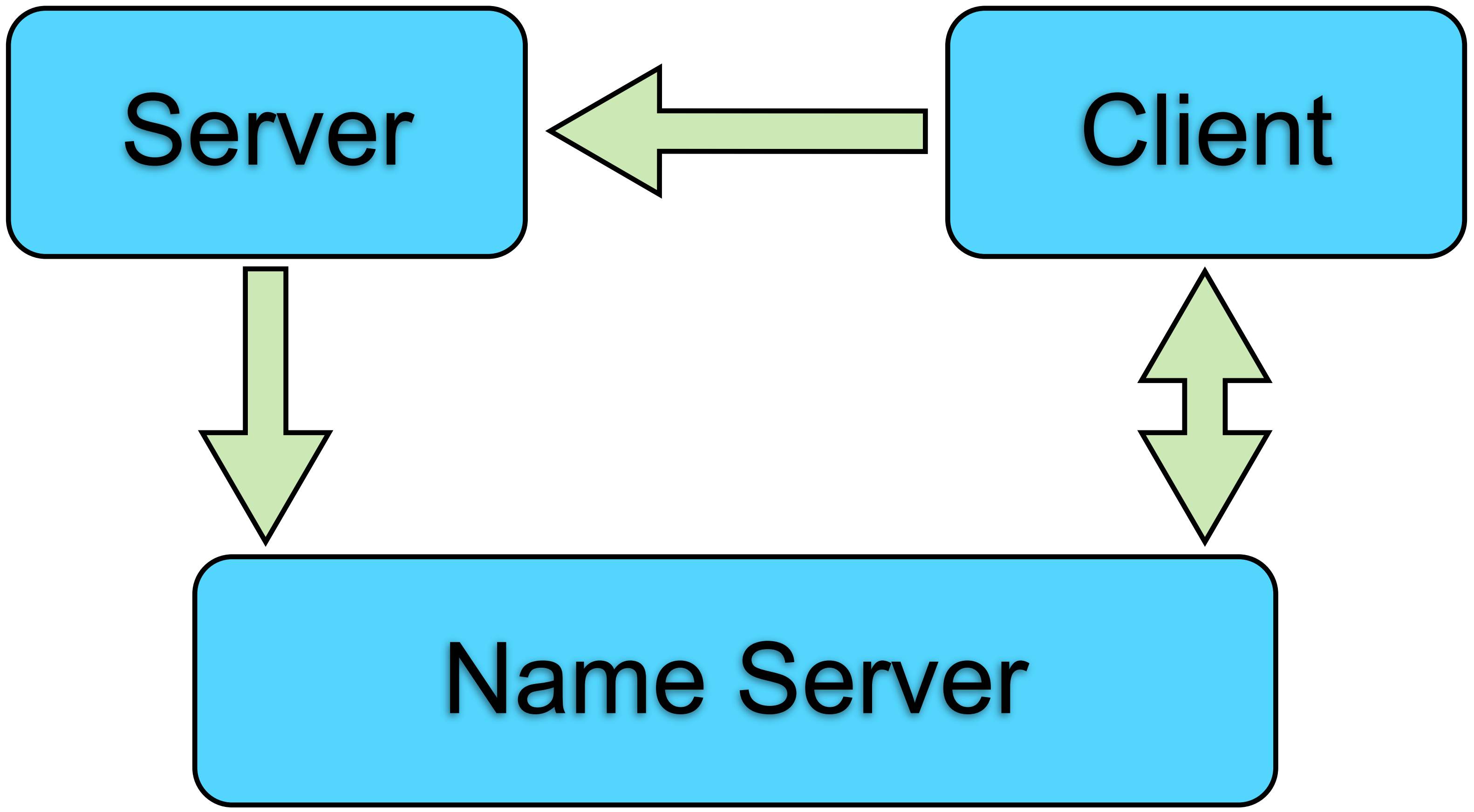
The pub-sub scenario with a separate broker is much more elaborate and allows to physically decouple communicating parties...

Pub-Sub with Broker



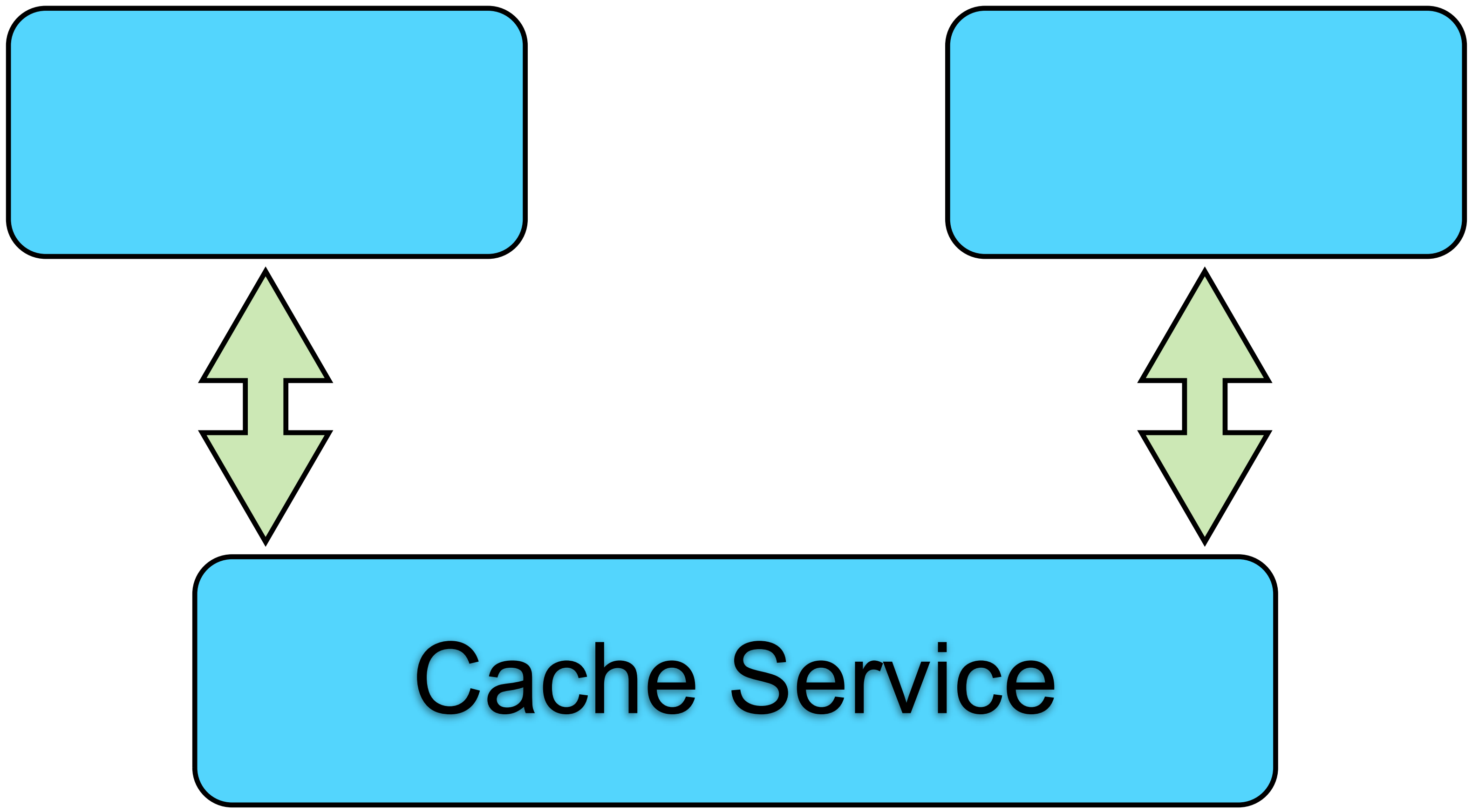
... This is useful for dynamically reconfigurable systems and to reduce processing overhead at the side of data source. Dedicated broker supports also a very flexible message matching scheme based on hierarchic tags.

Name Service



The name server can act as a “phone book” of the distributed system and allows other programs to find each other based on logical names instead of hard-coded network endpoints. This allows to reduce the set of addresses that all nodes have to be aware of.

Distributed Cache



A distributed cache complements the set of standard services and can act as a very simple shared data store. The cache allows to store complete parameters objects as well as arbitrary binary blobs. It is non-persistent.

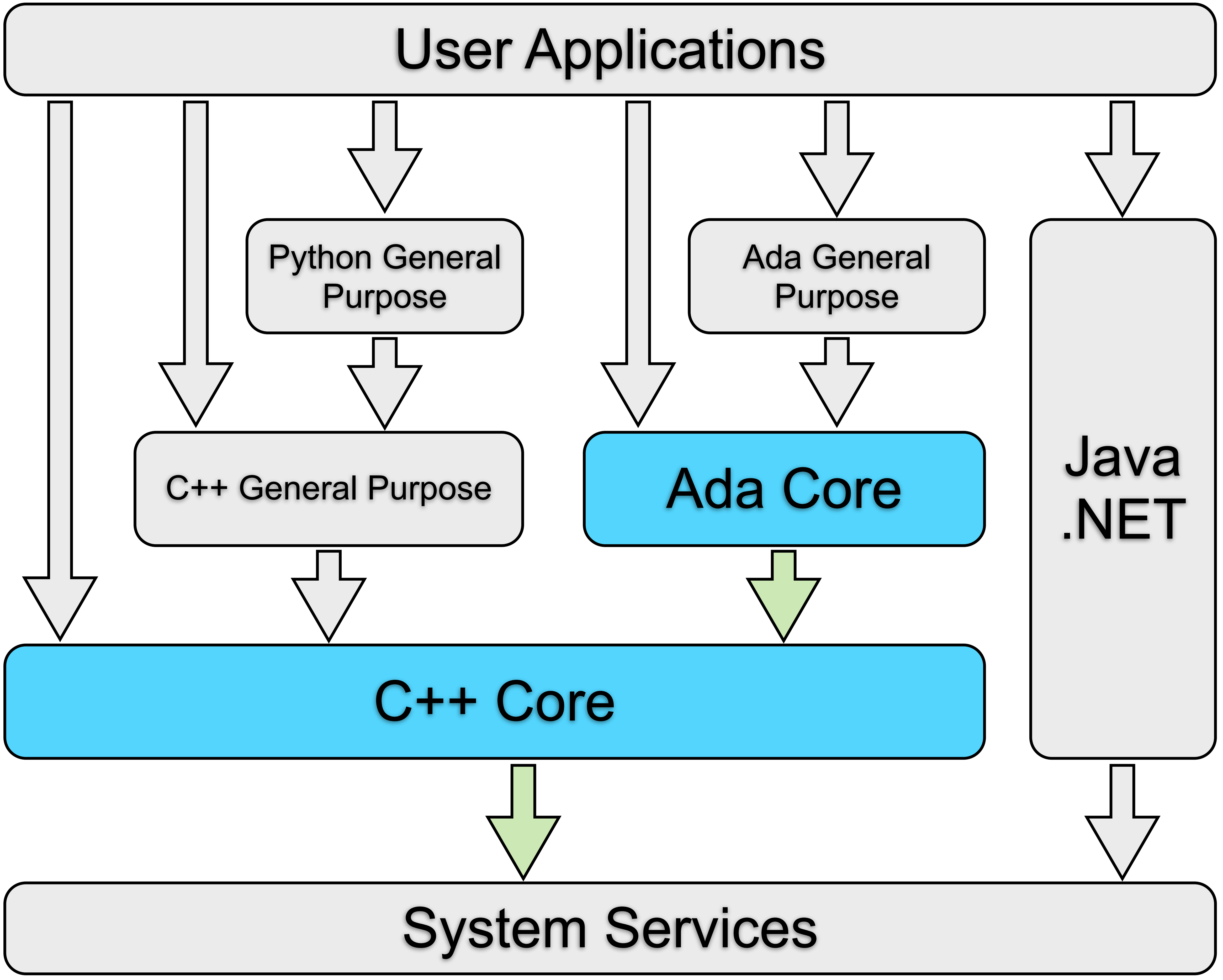
Concurrency Issues

Agent is task-safe, but...

Message handlers are executed in the context of agent's internal tasks - beware!

Beware the implicit multitasking that is introduced by agents via message handlers. With high-level general-purpose libraries all incoming messages are delivered in the context of agent's own tasks, so all references to shared program state should be properly handled.

Critical Systems



YAMI4 was built with critical systems in mind and its layered architecture reflects that - core library components implement only the most fundamental services and serve as “zones” of strict design and coding conventions. General-purpose components provide higher-level and more comfortable APIs, but at the cost of being less strict in their coding guidelines.

Core Library

Address the needs of critical systems:

- **minimized dependencies (do not neglect exotic platforms)**
- **strict coding rules and conventions**
- **minimal interface appropriate for restricted developments (eg. Ravenscar)**

Private Memory Partitions

Private memory partitions allow YAMI4 to work within the given block of memory and be isolated from the general-purpose allocator.

Both parameters objects and agents can work with private memory partitions.

The state of private allocator is reproducible.

Private memory partitions allow to isolate the communication aspect of the system from its other activity. Thanks to this, memory fragmentation or impossible to predict interferences from other program modules have no impact on distributed communication.

Private Memory Partitions

```
declare
   type Byte is mod 256;
   for Byte'Size use 8;

   type Byte_Buffer is
     array (Positive range<>) of Byte;

   One_Kilobyte : constant := 1024;

   Private_Memory :
     Byte_Buffer (1 .. One_Kilobyte);

   -- ...
```


In order to set up a private memory partition the user needs to allocate a contiguous memory block - this can be done in any way that is convenient in a particular context, but in critical systems an object created at library level will be the most likely choice.

Private Memory Partitions

```
declare
   Private_Memory :
     Byte_Buffer (1 .. One_Kilobyte);

   Content : Parameters_Collection :=
     Make_Parameters (Private_Memory'Address,
                      Private_Memory'Size / 8);

begin
   Content.Set_String ("name", "Maciej");
   -- ...
end;
```


A dedicated memory block is provided when the parameters object or the agent are constructed. Note: do not share a single memory block across many entities at the same time!

YAMI4 and Ravenscar

YAMI4 and Ravenscar

Related Ravenscar cornerstones:

- **static and flat set of tasks**
- **static set of protected objects, which are "communication points"**
- **external events mapped to protected procedures**

The Ravenscar profile defines several key restrictions and design guidelines that help build more predictable systems. These restrictions can be met in a distributed system with appropriate provisions at the level of communication layer.

YAMI4 and Ravenscar

Incoming messages in YAMI4 play the same role as interrupts in other systems.

Design the program structure in a way that follows this analogy.

From the YAMI4 point of view, the key idea is that incoming messages are very similar in nature to interrupts - both represent an external stimuli that has to be delivered somehow to the program and integrated with the rest of its code. The Ravenscar profile suggests to attach interrupts to protected procedures and this can be a valid strategy for incoming messages as well.

YAMI4 and Ravenscar

Guidelines for designers:

- **agent(s) at library level**
- **I/O activity driven by any top-level task, perhaps from the main loop**
- **message handlers as protected objects implementing the handler interface, at library level**

These guidelines are fully compatible with the Ravenscar profile. Core agents do not create their own internal tasks and instead rely on the user to provide appropriate execution resources. This can be done easily with all entities instantiated at library level, which additionally prevents dangling references between all wired components.

Message Handler Skeleton

```
protected type My_Handler_Type is
   new Message_Handler with

   procedure Call
      (Msg : in out Incoming_Message'Class);

   entry Get_Data (D : out Data_Type);

private
   -- ...
   Data_Ready : Boolean := False;
end My_Handler_Type;
```


A sketch of message handler - the Call procedure is a primitive operation of the Message_Handler interface and will be automatically called by the agent for each incoming message. This is analogous to attaching protected procedures as interrupt handlers.

Message Handler Skeleton

```
protected body My_Handler_Type is

   procedure Call
      (Msg : in out Incoming_Message'Class) is
   begin
      -- extract payload and store
      -- ...
      Data_Ready := True;
   end Call;

   -- ...

end My_Handler_Type;
```


The Call procedure is supposed to extract the payload data from the message and signal the availability of data to other parts (perhaps other tasks) of the program. This procedure should not involve any time-consuming processing - if necessary, such processing should be performed by other, dedicated tasks.

Message Handler Skeleton

```
protected body My_Handler_Type is

   -- ...

   entry Get_Data (D : out Data_Type)
     when Data_Ready is
   begin
      D := ...
      Data_Ready := False;
   end Get_Data;

end My_Handler_Type;
```


In this sketch, Get_Data is a protected entry with simple barrier that allows to synchronize other tasks with the reception of incoming message - this is how incoming messages, treated as external stimuli, interact with other parts of the program.

Message Handler Usage

```
   -- at library level:

   My_Agent : Agent;

   My_Handler : aliased My_Handler_Type;

begin

   My_Agent.Register_Object
      ("my_object", My_Handler'Access);

   -- ...
```


According to the guidelines defined earlier, agent(s) and message handler(s) are instantiated at library level and wired together during initialization phase. Thanks to the fact that all entities are statically allocated and live till the end of the program, there is no risk of dangling references and mutual references are perfectly safe.

Main Loop

```
procedure My_Critical_System is
   -- ...
   Timeout : Duration := 1.0;
   Timed_Out : Boolean;
begin
   -- any initialization that is needed
   -- ...
   loop
      My_Agent.Do_Some_Work
        (Timeout, Timed_Out);
   end loop;
end My_Critical_System;
```


Since core agents do not have their own internal tasks, the execution resource has to be explicitly provided - this can be done from any task, perhaps from the main program loop, via repeated invocations of Do_Some_Work. This operation performs a single unit of communication work and, once the incoming message is complete, passes it to appropriate handler.

Thank you!

www.inspirel.com/yami4