Erlang as an Enabling Technology for Resilient General-Purpose Applications on Edge IoT Networks

Igor Kopestenski
ICTEAM Institute
UCLouvain
Louvain-la-Neuve, Belgium
igor.kopestenski@uclouvain.be

Peter Van Roy
ICTEAM Institute
UCLouvain
Louvain-la-Neuve, Belgium
peter.vanroy@uclouvain.be

Abstract

Edge computing is one of the key success factors for future Internet solutions that intend to support the ongoing IoT evolution. By offloading central areas using resources that are closer to clients, providers can offer reliable services with higher quality. But even industry standards are still lacking a valid solution for edge systems with actual sense-making capabilities when no preexisting infrastructure whatsoever is available. The current edge model involves a tight coupling with gateway devices and Internet access, even when autonomous ad hoc IoT networks could perform partial or even complete tasks correctly.

In our previous research efforts, we have introduced Achlys, an Erlang programming framework that takes advantage of the GRiSP embedded system capabilities in order to bring edge computing one step further. GRiSP is an embedded board that can easily be programmed directly in Erlang without requiring deep low level knowledge, which offers the extensive toolset of the Erlang ecosystem directly on bare metal hardware. We have been able to demonstrate that our framework allows building reliable applications on unreliable networks of unreliable GRiSP nodes with a very simple programming API. In this paper, we present how Erlang can successfully be used to address edge computing challenges directly on IoT sensor nodes, taking advantage of our existing framework. We display results of deployed distributed programs at the edge and examples of the unique advantage that is offered by Erlang higher-order and concurrent programming in order to achieve reliable general-purpose computing through Achlys.

Erlang '19, August 18, 2019, Berlin, Germany

ACM ISBN 978-1-4503-6810-0/19/08…$15.00
https://doi.org/10.1145/3331542.3342567

CCS Concepts • **Computer systems organization → Embedded software**; *Sensor networks*; Reliability; Fault-tolerant network topologies; • **Software and its engineering → Distributed programming languages**.

Keywords Edge Computing, Internet of Things, Distributed Computing

ACM Reference Format:
Igor Kopestenski and Peter Van Roy. 2019. Erlang as an Enabling Technology for Resilient General-Purpose Applications on Edge IoT Networks. In *Proceedings of the 18th ACM SIGPLAN International Workshop on Erlang (Erlang '19), August 18, 2019, Berlin, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3331542.3342567

1 Introduction

The term *Edge Computing* refers to a distributed computing paradigm that brings processing resources closer to end users than they are in traditional cloud computing infrastructures. This emerging concept is a necessary response to the exponential development of IoT, as the current centralized design would collapse under the pressure of 75 billion IoT devices expected by 2025 [33]. However, even standardized state-of-the-art reference architectures for edge computing still implicitly require that computing is done *near* the edge at best, but not *at* the edge itself [17, 20]. And Cloud Service Providers (CSPs) such as Microsoft, Google and Amazon tend to make identical assumptions in their edge solutions [6].

Our work in Achlys is to focus on enabling edge computing directly on IoT devices, at the furthermost part of networks. We want to allow autonomous distributed applications to run at the edge even without Internet access or CSP services. This brings multiple highly desirable characteristics such as near zero-configuration deployment, decreased management and maintenance complexity, lower cost and fast local decision-making ability.

Moreover, we also want Achlys to be easily interoperable with the regular edge and cloud services. Therefore our goal is to extend the current model of edge computing without sacrificing any of the existing features.

Our work is part of a much greater EU research project called **LightKone**[1]. Achlys is only a single component of

[1] https://lightkone.eu

the LightKone Reference Architecture for edge computing (LiRA). The vision proposed by LightKone is currently being defined in a whitepaper that will soon be released. Our tool is built using other LightKone software innovations, and our goal is to bring them to the extreme edge and reduce the complexity of developing general purpose IoT edge computing applications in Erlang.

To allow the computing to be done directly at the edge, Achlys provides a task-based computation model for edge applications, together with a resilient key/value store (based on Lasp) and a resilient communication layer (based on Partisan). The resilient storage and communication are needed to overcome the unreliability of edge networks. The tasks in the task model are stored in the key/value store. Achlys is the initial research prototype that we have presented as an enabler of non-trivial resilient edge computing directly on clusters of IoT nodes. We have implemented this first version on the GRiSP embedded system board. Each GRiSP has a 300 MHz ARM processor and 64 MB of RAM with a 8 GB MicroSD card. It is a particularly suitable choice as it is currently the only system that offers *bare metal* Erlang, on top of its support for Digilent Pmod[2] sensor and actuator modules. This means that accessing sensor data can be done immediately in Erlang on GRiSP, without any intermediary layer. Achlys is designed to run on an ad hoc network of GRiSP boards, and uses this network to provide distributed resilient computation, storage, and communication.

In this document we will first overview the current state of our framework to present its fundamental concepts, abilities and limitations. We continue with case studies of actual applications built on top of our platform and display the results of our experiments. Finally we finish with concluding remarks and an insight at the vast opportunities that remain open for our further research.

1.1 Contributions

The motivation that drives the development of Achlys is the current lack of edge computing solutions that do not somehow still depend on cloud computing services. To the best of our knowledge, none of the existing edge systems is specifically designed to shift the programming paradigm entirely so that no **gateway** is a mandatory part of the architecture. Using Achlys, it is possible to run Erlang programs on sensing and actuating devices, without any existing infrastructure and with nearly zero configuration. Our task model is built on top to provide a very simple way to develop edge computing applications in Erlang, in synergy with our hardware and software components.

In Achlys, tasks are abstractions that contain Erlang functions and metadata used for efficient propagation and reliable execution across an entire cluster of GRiSP boards. Therefore the Achlys task model not only allows computing at the edge, but also performing tasks that do not need to be known by any of the nodes before executing them. In fact, by accessing the task model API from any GRiSP node in the network, one can disseminate Erlang programs reliably and use the cluster as a general-purpose computation platform.

In this document, we put an emphasis on the contributions below. Contrary to our previous research, we focus on Erlang-specific aspects both in terms of implementation improvements and usage :

- Demonstration of distributed IoT edge computing functions delpoyed on GRiSP boards using the Achlys task model
- In contrast to our previous work, we now provide Erlang code that shows implementation and usage of Achlys for IoT edge computing
- Emphasis on the simplicity of IoT sensor data collection at Erlang level
- Detailed description of major improvements to the framework, mainly the task model
- Description of the revised task model with its backpressure mechanisms and simplified API.
- Proposal of a possible entirely BEAM-based end-to-end architecture pattern
- Presentation of key elements for future research and further improvements

2 Background

In our previous study we have shown how GRiSP's architecture compares to a traditional system [21]. The major difference is that no operating system is required for the Erlang VM to run on it. This is a major benefit as it makes the clusters of IoT nodes homogeneous, as they are all Erlang nodes that are able to cooperate directly thanks to their Wi-Fi antennas. The heterogeneousness of sensor and IoT networks is otherwise a prominent blocking factor for successful deployments of these systems [15, 27]. We have now made sufficient progress in our implementation to proceed to the next phase of evaluation. We are no longer discussing the feasibility of our approach. Instead, we can now focus on evaluating our solution by applying it to solve real-world problems.

This architectural advantage enables us to overcome one of the major difficulties of edge IoT application development. We can write distributed programs and deploy them on ad hoc, infrastructure-agnostic networks of GRiSP nodes as we would on regular computers. But in addition, we also have immediate access to sensor and actuator modules in Erlang through the GRiSP software stack. Thus, the rich set of OTP tools and the accompanying properties such as fault tolerance and scalability are available for building resilient IoT edge systems. OTP provides an abstraction layer that captures recurrent behavioral patterns through a set of **OTP behaviors** [11].

[2] https://wikipedia.org/wiki/Pmod_Interface

Since it is directly available on GRiSP boards, we can port existing Erlang/OTP libraries on these embedded systems as we would for applications in cloud environments without any sort of complex low level modification. The primary goal of Achlys is to enable reliability where it is currently not thought of as possible, namely the inherent unreliable nature of IoT devices. From the bottom up, we present how we combine LightKone individual innovations to form a custom compound for Achlys. Our first layer is the GRiSP board, that strengthens our framework directly starting from the hardware thanks to bare-metal Erlang features. Our approach is to apply several layers of reinforcement to Achlys by superposing multiple LightKone innovations on top of GRiSP before writing our program as the upmost component.

For that purpose, we rely on a LightKone innovation : the Lasp programming library [29]. We use it both as a distributed key/value store for IoT sensor data and a propagation tool for our generic task model. Lasp provides access to a wide range of Conflict-free Replicated Data Types (CRDTs), that ensure that conflicting operations on a same data entry are automatically handled using the underlying conflict resolution algorithm [10, 18, 31]. Consequently, Achlys clusters are able to preserve *strong eventual consistency* of data across nodes. This property implies that *convergence* and *monotonicity* always hold for the state of a replica. Thence, our network of unreliable IoT devices becomes adequate for distributed applications that require reliable decentralized storage.

Achlys also inherits from Lasp's distribution module, Partisan. Partisan is used to bypass the native Erlang distribution protocol to achieve lower latency and increased scalability [5, 28]. This is possible thanks to a combination of two hybrid gossip algorithms called HyParView and Plumtree [25, 26]. Hence, Achlys clusters can organize in more efficient ways than the default full-mesh topology when required. And depending on the context, it is also possible to easily modify the virtual topology cluster-wide at runtime without affecting the logic of the application.

3 Achlys : Design and Usage

The current version of Achlys[3] is a research prototype and demonstrates a set of capabilities with our current hardware and software stack. Nevertheless, there are various features that are in Achlys that make it possible to port the prototype version towards actual industrial deployments. These will be detailed further. The goal of the initial version is to experiment and validate the approach in a first phase, while engineering is needed in order to materialize the LightKone Reference Architecture (LiRA) vision of the *thin edge* with a larger number of IoT sensor nodes [9].

[3] https://ikopest.me

3.1 The Erlang VM

The Erlang/OTP tools are designed in a way that provides scalability and fault tolerance properties, and the technology is being actively developed by a very dedicated community and industrial actors such as WhatsApp. Hence there are regular improvements and additions of new functionalities that Achlys is able to take advantage of directly. The requirements for robust edge IoT systems are extremely similar to those of systems supported by the BEAM and Erlang/OTP. Fault tolerance, scalability and concurrency are crucial in our system and Erlang's history of successful implementations with those properties was a compelling sign of a perfect match. And GRiSP boards being able to run a 6 MB large Erlang VM with all those features and our application together, and on a bare-metal embedded system.

3.2 Network Configuration

Achlys nodes form wireless ad hoc peer-to-peer networks in order to communicate. Building reliable distributed systems on top of these networks requires automatic neighbor discovery and seamless formation of fault-tolerant clusters. Erlang/OTP's `inet_res` and `inet` can be used as a basis for zero-configuration networking. Algorithm 1 demonstrates how a name can be resolved by retrieving the records via the `inet_dns` internal module.

```
example_lookup(Name, Class, Type) ->
  case inet_res:resolve(Name, Class, Type) of
    {ok,Msg} -> [inet_dns:rr(RR, data)
    || RR <- inet_dns:msg(Msg, anlist),
      inet_dns:rr(RR, type) =:= Type,
      inet_dns:rr(RR, class) =:= Class];
  {error,_} ->
    []
end.
```

Algorithm 1. DNS lookup documented in Erlang

3.3 Automated Dynamic Clustering

Based on the functionality described in Section 3.2, we can for example implement zero-configuration networking by providing resolution information in `erl_inetrc`. If all Achlys nodes can connect to neighboring peers, we can use Partisan to ensure reliable cluster communication.

By relying on the Partisan distribution library instead of the native Erlang distribution, we can not only decouple the application from static configuration, but from the actual physical topology of the network as well. By leveraging the ability to build virtual overlays on top of *ad hoc* networks, we are able to achieve zero-configuration networking but also zero-configuration **clustering** that allows nodes to cooperate despite network failures. This results in a three-way decoupling as there is no need for infrastructure, static configuration nor hard dependency on physical topologies [28].

Moreover, there is yet another feature that is provided by the LightKone stack in Achlys making it possible to change the virtual overlay behavior **at runtime**. It brings the unique capability of seamlessly modifying how the distribution protocol behaves in order to achieve maximum efficiency w.r.t. the observed physical topology. And it is planned for Achlys to be refactored to enable even greater modularity by providing the ability to developers to plug their own distribution implementation instead of Partisan, pushing abstraction even further. Partisan itself is already designed in a very modular fashion, and implementing the identical API would suffice for a developer to be able to substitute the default communication library by his own.

The default behavior of Achlys nodes is designed to enable and refresh the clustering procedure. Nodes will periodically scan their surroundings to determine if any other is reachable. Cluster formation will occur if possible, data is replicated and will eventually be consistent on all nodes.

3.4 Task Model

In our recent work, we have introduced the general-purpose task model API that we are developing in the Achlys framework [21]. Using Lasp's reliable distributed key/value storage, we are able to disseminate Erlang higher-order functions across clusters of GRiSP embedded boards [13, 29]. This allows programmers to use Achlys as a tool in Erlang distributed applications to persist entire programs the same way as in a database in a very simple way. Table 1 shows the usage and description of the API functions. We can see that there are four main components in our definition of a task.

1. The name : an arbitrary atom that describes the task
2. The targets : either a list of `nodes`, or `all` meaning that the task should be executed on each node
3. The ExecType : stands for **Execution Type**, either `single` or `permanent`. The latter makes a task loop permanently inside a controlled process at a configurable frequency.
4. The function : the Erlang function containing the task's job

Then, the convergence guarantees provided by the underlying Lasp and Partisan libraries ensure that each node in the cluster will eventually be able to execute these new programs, making them generic and consistently reprogrammable at runtime. Thus, by combining the abilities of the GRiSP embedded system and Lasp and offering a simple API that leverages their features, we facilitate edge IoT application development in Erlang.

3.4.1 Task Model Worker

Instead of relying on CRDTs with remove operations, we can declare either *single execution* or *permanent* tasks. We have implemented the Achlys task worker module specifically for the purpose of handling local task processing. By developing a dedicated component in our framework, we can benefit from a much more fine-grained control on our execution flow. When a task needs to be executed, the worker module spawns a new process and assigns the task's function to it. With this approach, we are now able to define a custom process spawning wrapper for our tasks that extracts the Erlang function from the task data structure.

We have implemented our `spawn_task/1` wrapper by using the `erlang:spawn_opt/2` function to specify a set of parameters that apply a standard memory control environment on all tasks. This has noticeably increased the system fault tolerance by allowing immediate shutdown of processes with uncontrolled memory growth without going through garbage collection [24]. This method brings two other considerable benefits for our framework's task model :

Heap Size Since we have already made an extensive amount of deployments on GRiSP boards in order to determine what parameters were optimal for the current hardware, we have made a prototype that holds a good basis for future fine-tuning features as well. Our plan is to extend the features of Achlys to integrate configurable parameters for the task process spawning wrapper. In our previous experiments, our main concern was memory since it is the most constrained variable in our current stack.

To adress this issue our task model worker enforces that the maximum heap size of the processes spawned for tasks to be no greater than 64 KB on GRiSP. We do already take the host's word size into consideration using a call to Algorithm 6 as a coefficient in our function such that on 64-bit architectures the `max_heap_size` is automatically doubled to 128 KB. This also enables our plans for even more configurable behavior, since we will extend the achlys API to expose a range of values for the upper bound of memory allocation.

Heap Placement Having a control over the maximum heap size of task execution processes increases fault tolerance as an unexpectedly memory-intensive operation kills the task process. But it does not affect the rest of the application. We can take advantage of this resilience improvement by combining it with the `off_heap` message queue option when spawning our worker processes.

Task processes that send messages will always create heap fragments instead of allocating space on the process heap. In distributed applications on ad hoc IoT edge sensor networks, it is more likely to observe small bursts of messaging activity separated by larger time intervals. Thence it is important for Achlys to guarantee that nodes are able to run for long periods with a near-constant memory footprint if no tasks are performed. Providing high throughput is not the goal of our framework, while supporting low power edge computing is an essential objective for Achlys. Off heap allocation for messages is a bit more expensive than on heap, but done at lower frequencies it provides the most efficient solution

for our framework. We have added an Achlys module for system-wide cleaning, including garbage collection, so that task processes can avoid phases of unnecessarily large heap footprint after activity bursts. This also reduces contention risk on the main locks of the receiving process [24].

Intuitively, one could think that a *grow-only* CRDT would be a bad choice for tasks that could be removed from the working set once they have been executed. However, our experiments have lead us to a different conclusion. As documented above the specification of Algorithm 4, we can declare a task that spawns other concurrent processes anyway. This means that there would be no point in removing a task from the working set CRDT, which would require it to allow `remove` operations at a high computational price. These CRDTs have not been specifically designed to be optimal for a context like our task model, and introduce a significant overhead as their conflict resolution algorithms are more complex.

The ExecType argument can currently not provide a transient mode, a task is executed either once or permanently. Removing the task from the CRDT does not prevent nodes to keep executing it. But similar behavior can be created with a single execution task that specifies a function with several loops. However, this does not allow the worker to have full control over the load, since backpressure can be applied by spawning a process that executes a permanent task at a controlled frequency that can be based on any stress parameter. Meanwhile a single execution task could possibly overload the worker, as iterations are embedded inside a single process. Using Grow-only Counters and Sets, the transient behavior will be achieved in future releases using Lasp's monotonic read function. It will act as a distributed threshold after which the task will stop being executed.

3.4.2 Data Structure

A task is stored in a map structure following the type specification shown in Algorithm 2.

```
-type task_targets() :: [node()] | bitstring().
-type task_execution_type() :: bitstring().
-type task() :: #{name => atom(),
                  targets => task_targets(),
                  execution_type =>
    task_execution_type(),
                  function => function()}.
```

Algorithm 2. Task data structure

3.4.3 Higher-Order Functions

Higher-order functions offer almost unlimited possibilities for our task execution model. A crucial point for these functions is to be fully replicated such that every node is able to execute them. To address this challenge, we have implemented a `squadron leader` module in Achlys, that performs automatic clustering as described in Section 3.3 Functions executed on the boards can be added at a randomly chosen node and will propagate automatically to all others. Hence none of the nodes in the cluster knows the program that it will be instructed yet all are guaranteed to eventually attempt to execute it. Algorithm 3 demonstrates a possible function for temperature minima reaction among 5 nodes.

```
mintemp() ->
  fun() -> Id = {<<"temp">>, state_gset},
    {ok, {_Id, _Meta, _Type, _State}} =
        lasp:declare(Id, state_gset),

    %% Taking 5 samples of temperature
    %% at 5 second intervals
    L = lists:foldl(fun
        (Elem, AccIn) ->
            timer:sleep(5000),
            Temp = pmod_nav:read(acc, [
                out_temp]),
            Temp ++ AccIn
    end, [], lists:seq(1,5)),
    Min = lists:min(L),
    Name = node(), Pid = self(),
    lasp:update(Id, {add, {Min, Name}}, Pid),
    %% Auxiliary process for blocking call
    spawn(fun() ->
      lasp:read(Id, {cardinality, 5}),
      {ok, S} = lasp:query(Id),
      Fetched = sets:to_list(S),

      %% Find global minimum
      {Minimum, Node} =
          lists:min(Fetched),
      Self = node(),
      %% Only if the current node is the one
      %% associated with the lowest temperature
    ,
      %% change the color of both LEDs to blue.
      case Node =:= Self of
          true -> [ grisp_led:color(X, blue)
                || X <- [1,2] ];
          _ -> [ grisp_led:color(X, red)
                || X <- [1,2] ]
        end
      end), end.
```

Algorithm 3. Temperature minima function

Calling `mintemp/0` returns a function that declares a grow-only set CRDT variable that will contain temperature minima from all nodes of the cluster. In this example, nodes take five temperature measurements using the `Pmod_NAV` sensor module, and store the lowest value in the CRDT. With the `lasp:read/2` function we block a process until the treshold of five elements in the CRDT set is reached. Once the process resumes, we know that the current node sees at least five elements corresponding to minima of all samples of all nodes. Based on this data, we actuate the LEDs of all nodes in our cluster such that the coldest GRiSP board lights up in blue. By accessing the Erlang shell on any of the GRiSP boards

in the network, we can write a function like `mintemp/0` at runtime and propagate it on the entire cluster.

3.4.4 Task Model API

The `achlys_task_server` process is one of the two core elements that power the Achlys task model. It is started right after the boot procedure on each GRiSP board that runs an Achlys release. Once it is initialized we declare a CRDT variable that is identified using the same value on all nodes. In Section 3.4.3, we have introduced an example higher-order Erlang function for distributed temperature minima computation. Achlys exposes an API where users can pass such functions as arguments and let the task server handle their propagation.

```
%% @doc Shortcut function exposing the utility
%% function that can be used to pass
%% more readable arguments to create
%% a task model variable instead of binary
%% strings.
-spec declare(Name::atom()
    , Targets::[node()] | all
    , ExecType::single | permanent
    , Func::function()) -> task() | erlang:
    exception().
declare(Name, Targets, ExecType, Func) ->
    achlys_util:declare(Name, Targets, ExecType
    , Func).
```

Algorithm 4. Task declaration helper

Once the task is propagated by calling the `achlys:bite/1` function , the worker component of the task model will handle the function execution accordingly. In order to inspect the current view of the working set seen by a node, the function presented in Algorithm 5 can be used on each node at all times.

```
-spec get_all_tasks() -> [achlys:task()] | [].
get_all_tasks() -> {ok, Set} = lasp:query(?
    TASKS),
    sets:to_list(Set).
```

Algorithm 5. Current working set view

```
%% 65536 bytes of heap on 32-bit systems like
    GRiSP
%% with 4-byte words. A total of 16384 words
%% on 64-bit and 32-bit architectures
-define(MAX_HEAP_SIZE , (
erlang:system_info(wordsize) * 1024 * 16)).
```

Algorithm 6. Erlang VM host word size

The task execution mode determines wether a function is to be run once in a process that is monitored by the worker, or permanently in steps encapsulated in processes that are periodically restarted each time the function is finished. A key feature of the task model worker is its periodical mechanism. It enables lookup and execution of functions with a task history control and workload monitoring. Algorithm 7 shows the main responsible handler.

```
handle_info(periodical_lookup , State) ->
  Tasks = achlys_util:query(?TASKS),
  %% Spawning a process for each
  %% task not previously finished
  %% or already running
  L = [ {H, spawn_task(T)} || {T, H} <- Tasks
 , permanent_execution(T, H) =:= true
 , dict:is_key(H, State#state.tasks) =:= false
    ],
  %% Adding new task processes to
  %% monitoring structure
  NewDict = lists:foldl(fun
    (Elem, AccIn) ->
        {H, Proc} = Elem,
        dict:store(H, Proc, AccIn)
  end, State#state.tasks, L),
  %% Scheduling the next cycle
  %% and hibernate the worker process
  schedule_periodical_lookup(
    State#state.task_lookup_interval),
  {noreply
      , State#state{tasks = NewDict}
      , hibernate};
```

Algorithm 7. OTP `gen_server` handler for periodical task lookup and execution

4 Case Study : True Range Multilateration

Using `Pmod_MAXSONAR` proximity sensor modules, it is possible to implement a program that enables collaborative sensing in Erlang using 3 GRiSP boards and perform resilient trilateration of an object. More accurately, the notion of **True Range Multilateration (TRM)** can be employed to determine the exact location of an object on a two-dimensional plane [8]. This method is based on the geometric principle that a point that sits simultaneously on two overlapping circles can be located if the radius and center coordinates of both of them are known. This method is used in many domains, such as in military radio systems. A notable real-world application of TRM in is US armed forces' Joint Tactical Information Distribution System (JTIDS) [19, 35], and our first use case study is inspired by an attempt to implement something similar but on a smaller scale and directly at the edge.

We envision it as a possible way to incorporate efficient tactical information distribution directly between combat elements that would be implemented using our Erlang framework. Such an application could be useful in situations where ground forces are at risk due to hostile smoke screens reducing their visibility. The U.S. Army Research Lab is currently conducting research that points out the opportunity that Internet of Battlefield Things (IoBT) represents [2, 3, 22, 23]. But in contrast to the commercial solutions that are being developed by industry, IoBT applications must be able to

Table 1. Achlys API summary

Function	Description	Example call
`achlys:declare(Name, Targets, ExecType, Func) -> achlys:task() | erlang:exception().`	Declares a task called Name, for a list of destination nodes or all the cluster and that will run Func a single time or permanently	`T = achlys:declare(my_task , all, permanent, fun() ->ok end).`
`achlys:bite(Task :: achlys:task()) ->ok.`	Propagates task Task across a cluster	`ok = achlys:bite(T).`
`achlys:get_all_tasks() ->[achlys:task()].`	Returns a local view of all tasks in the working set CRDT	`[ #{execution_type =><<0>>, function =>#Fun<erl_eval.20.128620087>, name =>my_task, targets =><<0>>}] = achlys:get_all_tasks().`

remain highly resilient even in complete absence of infrastructure and be able to form self-sustaining edge clusters without any kind of gateway or permanent connection.

Therefore we study the feasibility of designing a resilient IoT application with Achlys on GRiSP boards, using a 3-node cluster in order to locate an object reliably even if visibility is insufficient. Also, since our framework also strives for maximum geniricity in order to follow LightKone's project principles we increase the scope of our use by adding the requirement that none of the three participating nodes contains the code necessary for performing TRM. By doing so, we analyse the capabilities of our prototype both in terms of genericness and fault tolerance properties. These properties are highly desirable for distributed IoT systems at the edge in the case of IoBT.

The Digilent Pmod_MAXSONAR module is a 2×2.5cm sized sensing device that provides ultrasonic proximity measurements with 1-inch (2.54cm) precision up to 254-inches in front of it (6.45m) [7]. It can be directly plugged into a GRiSP board and called from Erlang in order to fetch the current value measured by the sensor at a 20 Hz frequency. Therefore, if two nodes equipped with this module cover overlapping areas detect an object they can be viewed as centers of two circles whose radius equals the measured distance as shown in Figure 1. Consequently, if the separation S between them is known, the Erlang program running on GRiSP has sufficient information in order to calculate the location and detect movements of an object at point a $P(x, y)$ where x, y are given by :

$$x = \frac{r_1^2 - r_2^2 + S^2}{2S} \ , \ y = \pm\sqrt{r_1^2 - x^2} \tag{1}$$

In case a third node is also covers the area that comprises $P(x, y)$, the uncertainty about y introduced by ambiguous points $a1, a2, a3$ can easily be eliminated since only one value is common in all three results.

4.1 Experiment Results

We assess the adequacy of our framework for the computation of the position of an object by True Range Multilateration using two GRiSP boards that are each equipped with a Pmod_MAXSONAR sensor module. In order to obtain our results,

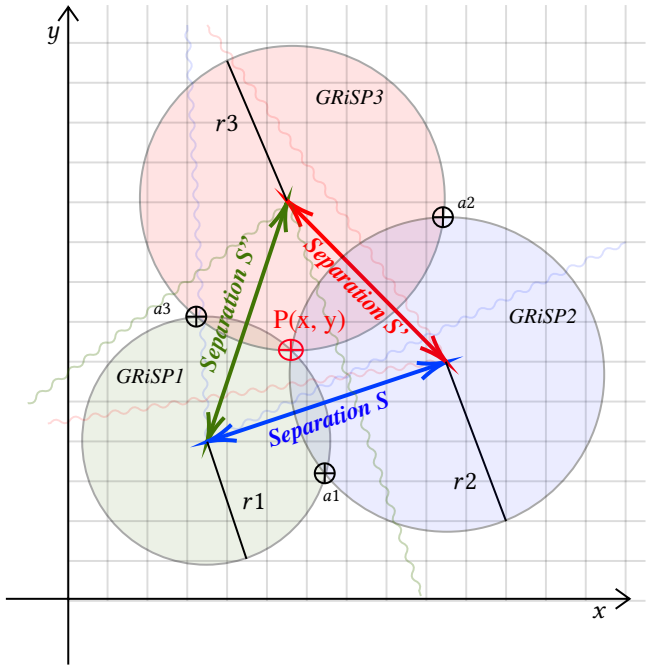

Figure 1. True Range Multirateration using Pmod_MAXSONARs on Achlys nodes

we provide the separation and target remote node parameters to a function that is fed to our task model. Algorithm 8 shows the function that we have implemented in order to translate Equation 1 on page 7 and that we disseminate using the `achlys:bite/1` API call.

Our first attempt allows us to assess the fit of our framework as a basis for more sophisticated TRM computing. In our experiments, we have been able to use the Achlys task model to calculate the coordinates of an object placed in front of the two GRiSP boards. The measurements were not precise enough and required manual adjustments in order to be reproduced. Nevertheless, in completely empty spaces and when MAXSONAR modules perceived no disturbances, Achlys' task model API was successful in permitting a TRM calculation.

```
-spec(multilateration(
    Separation :: pos_integer()
    , RemoteNode :: atom()) ->
    Fun :: erlang:function()).
multilateration(Separation, RemoteNode) ->
  fun() -> Node = node(),
    %% Local measurements CRDT
    Identifier =
        achlys_util:declare_node_crdt(sonar
        , state_gset),

    %% Updating every 10 seconds
    Poller = fun F() ->
      lasp:update(Identifier
      , {add, {erlang:monotonic_time()
          , pmod_maxsonar:get()}}
      , self()),
      timer:sleep(10000), F() end,
    spawn(fun() -> Poller() end),
    spawn(fun() ->
      %% Calculating result based on most
      %% recent measurements visible
      %% starting when at least one value
      %% is present on local and remote
      lasp:read(RemoteNode, {cardinality, 1}),
      {ok, RemoteSet} = lasp:query(RemoteNode),
      {ok, NodeSet} = lasp:query(Identifier),
      RemoteList = sets:to_list(RemoteSet),
      NodeList = sets:to_list(NodeSet),
      R1 = lists:max(NodeList),
      R2 = lists:max(RemoteList),
      R1Sq = math:pow(R1, 2),
      R2Sq = math:pow(R2, 2),
      S2 = 2 * Separation,
      SSq = math:pow(Separation, 2),
      X = ( R1Sq - R2Sq + SSq) / S2,
      Y = math:sqrt(R1Sq - math:pow(X, 2)),
      ResultId = achlys_util:declare_node_crdt(
    results
      , state_gset),

      lasp:update(ResultId
        , {add
          , {erlang:monotonic_time()
          , Node
          , {X, Y, (-Y)}}}
        , self())
    end)
  end.
```

Algorithm 8. True Range Multilateration task function

This assessment is particularly encouraging when considering the results of our second case study. The TRM computation is a minimal prototype of a distributed application that can already somehow relate to a much more complex state of the art military infrastructure such as the JTIDS/TADIL-J tactical communication system. IoBT distributed systems are especially hard to conceive due to the extremely hostile environments that they must be able to cope with [2]. This makes them non-trivial by nature and thus our case study

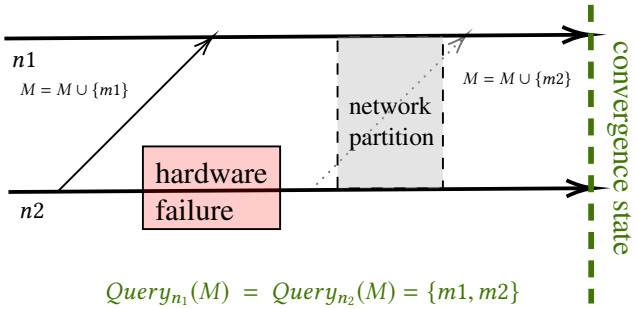

Figure 2. Deliberate failures by hardware reset and network partition

cannot cover such tasks alone, but our approach is a first correct step in that direction.

5 Case Study : Magnetic Field Surveillance

Our second use case is built using 2 GRiSP nodes and a laptop station. We will use the task model API to instruct one of the boards to sense the surrounding magnetic field periodically at fixed intervals. The GRiSP board is equipped with a `Pmod_NAV` sensor module and reads values from its 3-axis magnetometer.

For the purpose of this experiment, we state that these measurements must be stored in a set CRDT and replicated to ensure reliability. Then, we use the hardware reset button on the GRiSP board to simulate the most abrupt system failure that can occur. It will result in immediate ungraceful interruption of the running program and start rebooting the system. Figure 2 shows the timelines of the two nodes, where the red area indicates the interval during which the second board is restarting. No measurements have been persisted locally on the board, hence after successfully rebooting the node has lost all the previous measurements. But the replica on the first board will be repropagated ensuring that $m1$ is now visible again on both nodes.

The second phase after recovery is a network partition. This time the board is isolated far from the other peer and is not within radio transmission range anymore. The CRDT is updated with new measurements, as the task that we have added to the working set is declared as `permanent` meaning that it is executed indefinitely. We want our application to behave identically regardless of the temporary partition that prevents the propagation. Once the boards are brought back at communication distance, the cluster state should converge to the set of all measurements being visible on both nodes. The computer station is a spectating node that is used to display health status and data on nodes inside the cluster.

Fault tolerance is a cornerstone for designing reliable and distributed IoT programs at the edge. Therefore we have made it essential for Achlys nodes to work in presence of failures. This case study is a practical way to evaluate our

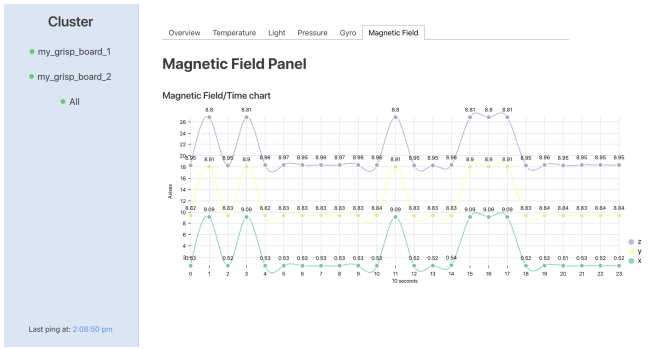

Figure 3. Cluster health and data live dashboard

task model in order to verify that it does not impede the fault tolerance properties of the rest of the components, and that it is reliable.

5.1 Experiment Results

Figure 3 shows a web client that displays the live dashboard that we were able to deploy successfully as a part of our previous work [13]. We deploy a two-node cluster and use it to show how redundancy allows us to store data reliably. Our Elixir web server can easily include Lasp as a dependency, making it fully interoperable with an Achlys cluster. By reading the data stored in the CRDT variable that contains the measurements provided by `my_grisp_board_2`, we see that no information has been lost despite a hard reset made on the GRiSP board at runtime. The replica stored on `my_grisp_board_1` is propagated again and can be displayed seamlessly.

The web client running on the laptop has been written as a separate NodeJS application. It calls the Elixir web server at 10 second intervals and plots magnetometer measures from our GRiSP board. This initial version leads to an interesting result. From an architectural standpoint, we now have a NodeJS component and JavaScript dependencies such as ReactJS in addition to our Achlys code base and Elixir server. But the recent release of Phoenix LiveView[4] allows us to discern a new design pattern thanks to our experiment.

Since LiveView enables rendering HTML on Elixir servers, we can substitute our JavaScript client by a LiveView implementation. And since GRiSP is bare metal Erlang, there is not a single remaining dependency that does not run on the BEAM anymore. We can easily extract a pattern for a full edge-to-cloud deployment architecture that shows the simplicity of integrating Achlys edge computing applications. Figure 4 shows how we can design infrastructures that combine Achlys clusters with the AntidoteDB[5] Erlang distributed database.

[4] https://github.com/phoenixframework/phoenix_live_view
[5] https://antidotedb.eu

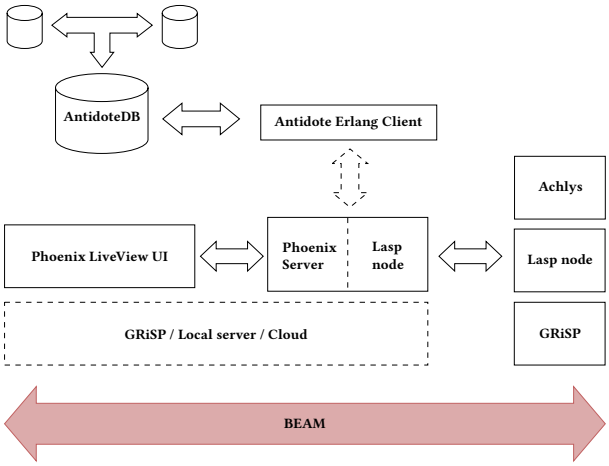

Figure 4. BEAM-enabled end-to-end architecture pattern.

6 Related Work

In this section, we provide an overview of some industry solutions in the *edge* computing landscape.

6.1 Non-Trivial Algorithms on IoT Devices

Learning algorithms have already been successfully ported to resource constrained devices without the need for an Internet connection. For instance, Long Short Term Memory (LSTM) deep learning models have been used to develop breathing acoustic based authentication with an accuracy level around 90% with a memory footprint below 200 KB [14]. The TenSense [32] node is an Industrial IoT sensor node with critical reliability requirements that allows monitoring the structural health of large compound infrastructures such as bridges with the help of smart tension sensing nodes. Both of these studies display valid results of complex computations done directly on sensor nodes.

6.2 Multi-access Edge Computing

Formerly known as Mobile Edge Computing, Multi-access Edge Computing (MEC) is the ETSI standard architecture for provisioning latency-sensitive applications [17]. The MEC approach is strongly catalyzed by network virtualization [30, 34] and accelerated hardware such as high performance FGPA-based SmartNICs [1]. But these innovations are used by network providers in order to allow the IoT sensor dataflow to reach the cloud computing providers faster, and do not consider sensor nodes in their designs.

6.3 EdgeX Foundry

The EdgeX Foundry [4] is a software platform for programming IoT applications at the edge. Its main characteristics are being vendor-neutral, OS-agnostic and hardware-agnostic with the main objective of increasing interoperability among the highly heterogenous landscape of the IoT edge. But a deeper look into the reference architecture, implementation

and requirements show that EdgeX does not actually fulfill these conditions. Similarly to the techniques detailed in Section 6.2, complex sense-making tasks are still done by sending IoT sensor data through a gateway towards what is labelled as "additional services". These actually refer to traditional third-party cloud features that perform the necessary computations.

6.4 IoT Azure RA

The IoT Azure Reference Architecture describes the guidelines for Azure IoT solution implementation. When compared to the LightKone Reference Architecture (LiRA), the Azure RA fails to address the D2D horizontal collaboration at the edge [6]. In fact, the IoT layer is only able to communicate vertically with a gateway towards the Azure cloud infrastructure. This makes it entirely dependent on this access point and shows that it is a single point of failure IoT Hub.

6.5 Amazon Greengrass RA

AWS Greengrass is Amazon's equivalent of Microsoft's Azure RA described in Section 6.4. This IoT solution still requires a cloud access for management, analytics and durable storage [6]. But it does overcome some of the shortcomings of the fully connection-dependent Azure IoT product. D2D communication and machine learning inference are two examples of features enabled by AWS IoT Greengrass.

6.6 Observations

We can easily highlight the lack of actual computing at the edge in these architectures. A pattern emerges in the approaches described above and Achlys is fundamentally different : our work complements and extends the current edge programming paradigm. We focus on showing that edge computing without gateways is possible as well. To the best of our knowledge, there are currently no systems that offer real fault tolerant generic edge computing on autonomous IoT networks without a permanent infrastructure access. The current programming paradigm for the edge is *dogmatic* : the belief that IoT edge networks can not perform sense-making i.e. complex decision tasks is so strong that such distributed system designs are nonexistent.

7 Future Research

We are currently implementing, testing and documenting components in Achlys in order to improve the overall quality of our framework. The RTEMS[6] library that is bundled with the cross-compiled Erlang VM into a single release loaded by the GRiSP bootloader currently induces a memory allocation mismatch at runtime. The resulting consequence is that the Erlang VM cannot use all of the available 64MB of memory on a GRiSP board. Peer Stritzinger, designer of GRiSP, has brought to our attention that this can be better understood if the RTEMS shell is accessed during execution in order to inspect memory allocation and potential issues. Therefore this will be part of our work in order to overcome Achlys' current limitations.

We have been able to significantly optimize the memory usage thanks to a monotonic approach in the entire program. With these improvements, Achlys is now much more stable using state based CRDTs stored in memory and the default Partisan peer service. The Lasp storage module that we were using to write on the MicroSD cards is not always able to recover from failures such as hard resets on GRiSP, even if it works on computers. Once the boards attempt to reboot, if the persistence operations were not properly terminated, the file is corrupt and the system crashes. Instead, we will now use Erlang's built-in memory-to-disk and vice- versa functions in Achlys to minimize the risk and potential loss of data in case of failure.

We have observed that there are far more benefits in adopting the grow-only monotonicity approach wherever it is applicable, as the conflict resolution is trivial, while the performance has greatly improved and the scope of functionalities has remained the same.

In the future, we will continue our work in this direction by trying to limit the presence of "happened-before" relations between distributed operations. Reducing the need for causal ordering during the design of a system by abstracting time itself makes it much easier to reason about the actual logic. This will be the preferred approach in our future work, as the system model assumptions that are made for operation-based CRDTs or δ-CRDTs are not always applicable on the IoT edge. For example, a reliable causal communication or a reliable local persistence mean on each node is not possible nor desired in our case [12].

Instead, our future improvements will be oriented towards a solution where the worst conditions are always assumed. This way we target the issues that are specific to edge sensor nodes and address them properly. The Erlang/OTP version 22.0 was released on May 14th 2019 and introduces numerous changes that will possibly greatly improve the overall performance, reliability and scalability of Achlys. Examples include the support fragmentation of large payloads in the native Erlang distribution protocol.

The GRiSP software stack now supports Erlang **N**ative **I**mplemented **F**unctions. These are functions that can be implemented in C for example, and called at application level directly in Erlang. It is also not limited to C, there are emerging combinations such as Rust NIFs combined with Erlang programs to achieve high performance for pure computations before going back to regular Erlang execution level.

And not only is it possible to incorporate different languages in Erlang programs, but it is also possible to run

[6] https://rtems.org

Erlang, and therefore Achlys, on any platform. The cornerstone of a sustainable edge computing and cloud computing architecture is its ability to handle heterogeneous devices in extremely high numbers. From Erlang directly on bare-metal to CSP datacenters, LightKone would offer the first solution that is entirely decoupled from any dependencies as a single technology could be used to bring harmony across vertical and horizontal layers of the Cloud.

Memory-wise, some significant enhancements in allocator behaviors could allow a substantial decrease of memory waste. A particularly interesting case for Achlys would be to take advantage of the `madvise(2) + MADV_FREE` system call that is now made by the emulator to advise the OS to free and reclaim an unused memory area [16]. Also, the ETS storage module has seen its scalability improved, and could represent a direct gain for the Lasp ETS storage backend. These are only few among the highly desirable improvements that we intend to leverage in future releases of the Achlys framework. Future Proof-of-Concept deployments at larger scales include :

- LightKone Smart Agriculture system using hygrometry sensor modules and actuators and the Achlys task model to control a subsurface irrigation pipeline in order to increase production efficiency autonomously and reliably.
- A second version of the GRiSP hardware is already being developed, and our partnership with Peer Stritzinger GmbH will allow us to integrate next-generation embedded Erlang systems into our deployments. Both Ethernet and Wi-Fi connectivity will be available on the second model, and will come with a nearly roughly times more efficient system. This will enable 4G and 5G connectivity through Android phones easily. Android smartphones can easily tether Internet connectivity through USB towards an Ethernet port. Since the GRiSP2 boards will have a slot for a Lithium battery the boards will remain powered on without a permanent source of power, and be charged at runtime.

8 Conclusion

In this paper we introduce a novel approach for reliable edge computing on IoT sensor networks. Our prototype aims at extending the current model of edge computing to the *actual edge* using Erlang. We have implemented Achlys, our prototype framework written in Erlang that is designed to run on ad hoc networks of GRiSP embedded systems. We have successfully ported the Lasp and Partisan libraries directly on the IoT edge. Using CRDTs, we are able to provide reliable distributed storage. In addition, we propose a generic platform that offers a task model exposing a straightforward API that allows clusters to execute programs in a resilient way even if they had no information about these tasks at compilation time.

We have evaluated our prototype on its fault tolerance, genericness and interoperability by showing how two real world non-trivial edge IoT use cases can be solved using Achlys. The positive results of our case studies lead to two conclusions. First, the current edge computing model can in fact be extended in order to reach the furthermost border of the network edge, at the sensor and actuator layer. Secondly, we demonstrate that Erlang can efficiently support this new model.

Contrary to current "edge computing" industry products, our prototype does not require gateways or other infrastructure in order to perform sensor data processing. The GRiSP platform offers Erlang's out-of-the-box concurrency and fault tolerance properties directly on bare metal. These properties have been used in order to support telecommunication systems and complex network orchestration services with very high resilience despite high failure and attrition rates. With Achlys, we intend to also take advantage of the highly interoperable nature of Erlang. In our future work, we will further develop Achlys in order to combine both data center infrastructures and autonomous edge networks in a BEAM-only unified architecture.

Acknowledgments

This work is partially funded by the LightKone European H2020 Project under Grant Agreement No. 732505. The authors would like to thank Giorgos Kostopoulos of Gluk Advice BV for information on precision agriculture, Peer Stritzinger from Peer Stritzinger GmbH for support and advice for GRiSP programming, the anonymous reviewers whose detailed comments helped to greatly improve the article and finally Joe Armstrong, the father of Erlang, for all of his work.

References

[1] 2018. FPGA SmartNICs with Acceleration for NFV. http://www.ethernitynet.com/products/smartnics/.
[2] 2018. Internet of Battlefield Things (IOBT) | U.S. Army Research Laboratory. https://www.arl.army.mil/www/default.cfm?page=3050.
[3] 2018. What Is the Internet of Military or Battlefield Things, or IoMT and IoBT? https://publications.computer.org/cloud-computing/2018/03/22/internet-of-military-battlefield-things-iomt-iobt/.
[4] 2019. 2. Introduction — EdgeX Documentation. https://docs.edgexfoundry.org/Ch-Intro.html#edgex-foundry-service-layers.
[5] 2019. Agner: Rethinking the Distributed Actor Runtime For Greater Scalability. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/atc19/presentation/meiklejohn
[6] 2019. LightKone Initial Runtime Edge Computing System. https://wordix.inesctec.pt/wp-lightkone/wp-content/uploads/2019/04/D3.1_InitialRuntimeEdgeComputingSystem.pdf.
[7] 2019. Pmod MAXSONAR: Maxbotix Ultrasonic Range Finder. https://store.digilentinc.com/pmodmaxsonar-maxbotix-ultrasonic-range-finder/.
[8] 2019. True Range Multilateration. *Wikipedia* (April 2019). https://en.wikipedia.org/w/index.php?title=True_range_multilateration&oldid=892448265.

[9] Ali Shoker, Peer Stritzinger, and Giorgos Kostopoulos. 2019. LiRA : LightKone Reference Architecture Presentation. https://github.com/achlysproject/achlys/blob/master/resources/LiRA.pdf.

[10] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2014. Efficient State-Based CRDTs by Delta-Mutation. *arXiv:1410.2803 [cs]* (Oct. 2014). arXiv:cs/1410.2803 http://arxiv.org/abs/1410.2803.

[11] Joe Armstrong. 2007. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, Raleigh, N.C. OCLC: 141384617.

[12] Angelos Bilas, Jesus Carretero, Toni Cortes, Javier Garcia-Blas, Pilar González-Férez, Anastasios Papagiannis, Anna Queralt, Fabrizio Marozzo, Giorgos Saloustros, Ali Shoker, Domenico Talia, and Paolo Trunfio. 2019. Data Management Techniques. *Ultrascale Computing Systems* (Jan. 2019), 85–126. https://doi.org/10.1049/PBPC024E_ch4

[13] Alexandre Carlier, Igor Kopestenski, and Dan Martens. 2018. Lasp on Grisp : Implementation and Evaluation of a General Purpose Edge Computing System for Internet of Things.

[14] Jagmohan Chauhan, Jathushan Rajasegaran, Suranga Seneviratne, Archan Misra, Aruna Seneviratne, and Youngki Lee. 2018. Performance Characterization of Deep Learning Models for Breathing-Based Authentication on Resource-Constrained Devices. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 4 (Dec. 2018), 1–24. https://doi.org/10.1145/3287036

[15] M. Du, K. Wang, Y. Chen, X. Wang, and Y. Sun. 2018. Big Data Privacy Preserving in Multi-Access Edge Computing for Heterogeneous Internet of Things. *IEEE Communications Magazine* 56, 8 (Aug. 2018), 62–67. https://doi.org/10.1109/MCOM.2018.1701148

[16] Erlang. 2019. Erlang Runtime System Release Notes for Versions of OTP 22.0 and ERTS 10.4. http://erlang.org/doc/apps/erts/notes.html#erts-10.4.

[17] ISG ETSI MEC. 2019. Multi-Access Edge Computing (MEC): Framework and Reference Architecture.

[18] Viktória Fördős and Francesco Cesarini. 2016. CRDTs for the Configuration of Distributed Erlang Systems. In *Proceedings of the 15th International Workshop on Erlang (Erlang 2016)*. ACM, New York, NY, USA, 42–53. https://doi.org/10.1145/2975969.2975974

[19] W.R. Fried. 1978. Principles and Simulation of JTIDS Relative Navigation. *IEEE Trans. Aerospace Electron. Systems* AES-14, 1 (Jan. 1978), 76–84. https://doi.org/10.1109/TAES.1978.308581

[20] Wazir Zada Khan, Ejaz Ahmed, Saqib Hakak, Ibrar Yaqoob, and Arif Ahmed. 2019. Edge Computing: A Survey. *Future Generation Computer Systems* 97 (Aug. 2019), 219–235. https://doi.org/10.1016/j.future.2019.02.050

[21] Igor Kopestenski and Peter Van Roy. 2019. Achlys: Towards a Framework for Distributed Storage and Generic Computing Applications for Wireless IoT Edge Networks with Lasp on GRiSP. In *SmartEdge'19 - The Third International Workshop on Smart Edge Computing and Networking (SmartEdge'19)*. Kyoto, Japan.

[22] Alexander Kott. 2018. Challenges and Characteristics of Intelligent Autonomy for Internet of Battle Things in Highly Adversarial Environments. *arXiv:1803.11256 [cs]* (March 2018). arXiv:cs/1803.11256 http://arxiv.org/abs/1803.11256.

[23] Alexander Kott, Ananthram Swami, and Bruce J. West. 2016. The Internet of Battle Things. *Computer* 49, 12 (Dec. 2016), 70–75. https://doi.org/10.1109/MC.2016.355

[24] Lukas Larsson. Wed Apr 27 15:12:39 CEST 2016. [Erlang-Questions] Max Heap Size. http://erlang.org/pipermail/erlang-questions/2016-April/088965.html.

[25] Joao Leitao, Jose Pereira, and Luis Rodrigues. 2007. Epidemic Broadcast Trees. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems (SRDS '07)*. IEEE Computer Society, Washington, DC, USA, 301–310. http://dl.acm.org/citation.cfm?id=1308172.1308243.

[26] Joao Leitao, Jose Pereira, and Luis Rodrigues. 2007. HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '07)*. IEEE Computer Society, Washington, DC, USA, 419–429. https://doi.org/10.1109/DSN.2007.56

[27] J. Lin, W. Yu, N. Zhang, X. Yang, H. Zhang, and W. Zhao. 2017. A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications. *IEEE Internet of Things Journal* 4, 5 (Oct. 2017), 1125–1142. https://doi.org/10.1109/JIOT.2017.2683200

[28] Christopher Meiklejohn and Heather Miller. 2018. Partisan: Enabling Cloud-Scale Erlang Applications. *arXiv:1802.02652 [cs]* (Feb. 2018). arXiv:cs/1802.02652 http://arxiv.org/abs/1802.02652.

[29] Christopher Meiklejohn and Peter Van Roy. 2015. Lasp: A Language for Distributed, Coordination-Free Programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming (PPDP '15)*. ACM, New York, NY, USA, 184–195. https://doi.org/10.1145/2790449.2790525

[30] R. Muñoz, R. Vilalta, N. Yoshikane, R. Casellas, R. Martínez, T. Tsuritani, and I. Morita. 2018. Integration of IoT, Transport SDN, and Edge/Cloud Computing for Dynamic Distribution of IoT Analytics and Efficient Use of Network Resources. *Journal of Lightwave Technology* 36, 7 (April 2018), 1420–1428. https://doi.org/10.1109/JLT.2018.2800660

[31] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Vol. 6976. Springer Berlin Heidelberg, Berlin, Heidelberg, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29 http://link.springer.com/10.1007/978-3-642-24550-3_29.

[32] Michail Sidorov, Nhut Phan Viet, Atsushi Okubo, Yukihiro Matsumoto, and Ren Ohmura. 2019. TenSense: IIoT Enabled Sensor Node for Remote Measurement of a Bolted Joint Tension. (2019), 10.

[33] Statista. 2018. IoT: Number of Connected Devices Worldwide 2012-2025. https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/.

[34] Mostafa Uddin, Tamer Nadeem, and Santosh Kumar Nukavarapu. 2019. Extreme SDN Framework for IoT and Mobile Applications Flexible Privacy at the Edge. *IEEE International Conference on Pervasive Computing and Communications* (2019), 11.

[35] William S. Widnall, Giuseppe F. Gobbini, and John F. Kelley. 1982. *Decentralized Relative Navigation and JTIDS/GPS/INS Integrated Navigation Systems:*. Technical Report. Defense Technical Information Center, Fort Belvoir, VA. https://doi.org/10.21236/ADA116417 http://www.dtic.mil/docs/citations/ADA116417.