COINRG
Internet-Draft
Intended status: Experimental
Expires: 9 February 2024

D. Kutscher
HKUST(GZ)
T. Kaerkkaeinen
J. Ott
Technical University Muenchen

8 August 2023

Directions for Computing in the Network
draft-irtf-coinrg-dir-00

Abstract

In-network computing can be conceived in many different ways -- from active networking, data plane programmability, running virtualized functions, service chaining, to distributed computing.

This memo proposes a particular direction for Computing in the Networking (COIN) research and lists suggested research challenges.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 9 February 2024.

Table of Contents

1. Introduction

Recent advances in platform virtualization, link layer technologies and data plane programmability have led to a growing set of use cases where computation near users or data consuming applications is needed -- for example, for addressing minimal latency requirements for compute-intensive interactive applications (networked Augmented Reality, AR), for addressing privacy sensitivity (avoiding raw data copies outside a perimeter by processing data locally), and for speeding up distributed computation by putting computation at convenient places in a network topology.

In-network computing has mainly been perceived in five variants so far: 1) Active Networking [ACTIVE], adapting the per-hop-behavior of network elements with respect to packets in flows, 2) Edge Computing as an extension of virtual-machine (VM) based platform-as-a-service, 3) programming the data plane of SDN switches (through powerful programmable CPUs and programming abstractions, such as P4 [SAPIO]), 4) application-layer data processing frameworks, and 5) Service Function Chaining (SFC).

Active Networking has not found much deployment in the past due to its problematic security properties and complexity.

Programmable data planes can be used in data centers with uniform infrastructure, good control over the infrastructure, and the feasibility of centralized control over function placement and scheduling. Due to the still limited, packet-based programmability model, most applications today are point solutions that can demonstrate benefits for particular optimizations, however, often without addressing transport protocol services or data security that would be required for most applications running in shared infrastructure today.

Edge Computing (in the ETSI Multi-access Edge Computing [MEC] variant, as traditional cloud computing) has a fairly coarse-grained (VM-based) computation-model and is hence typically deploying centralized positioning/scheduling though virtual infrastructure management (VIM) systems. Besides such industry-driven activities, manifold research approaches to edge computing with varying granularity and orchestration approaches, among other differiating elements, have been pursued [EDGESURVEY] [FOGEDGE].

Microservices can be seen as a (lightweight) extension of the cloud computing model (application logic in containers and orchestrators for resource allocation and other management functions), leveraging more lightweight platforms and fine-grained functions. Compared to traditional VM-based systems, microservice platforms typically employ a "stateless" approach, where the service/application state is not tied to the compute platform, thus achieving fault tolerance with respect to compute platform/process failures.

Application-layer data processing such as Apache Flink [FLINK] provide attractive dataflow programming models for event-based stream processing and light-weight fault-tolerance mechanisms -- however systems such as Flink are not designed for dynamic scheduling of compute functions.

Modern distributed applications frameworks such as Ray [RAY], Sparrow [SPARROW] or Canary [CANARY] are more flexible in this regard -- but since they are conceived as application-layer frameworks, their scheduling logic can only operate with coarse-granular cost information. For example, application-layer frameworks in general, can only infer network performance, anomalies, optimization potential indirectly (through observed performance or failure), so most scheduling decisions are based on metrics such as platform load.

Service Function Chaining (SFC, [RFC7665]) is about establishing IP tunnels between processing functions that are expected to work on packets or flows -- for applications such as inspection and classification, so that some of these functions could be seen as elements in a COIN context as well.

2. Terminology

We are using the following terms in this memo:

Program: a set of computations requested by a user

Program Instance: one currently executing instance of a program

Function: a specific computation that can be invoked as part of a program

Execution Platform: a specific host platform that can run function code

Execution Environment: a class of target environments (execution platforms) for function execution, for example, a JVM-based execution environment that can run functions represented in JVM byte code

3. Computing in the Network vs Networked Computing vs Packet Processing

Many applications that might intuitively be characterized as "computing in the network" are actually either about connecting compute nodes/processes or about IP packet processing in fairly traditional ways.

Here, we try to contrast these existing and widely successful systems (that probably do not require new research) with a more novel "computing in the network (COIN)" approach that revisits the function split between computing and networking.

3.1. Networked Computing

Networked Computing exists in various facets today (as described in the Introduction). Fundamentally, these systems make use of networking to connect compute instances -- be it VMs, containers, processes or other forms of distributed computing instances.

There are established frameworks for connecting these instances, from general purpose Remote Method Invocation/Remote Procedure Calls to system-specific application-layer protocols. With that, these systems are not actually realizing "computing in the network" -- they are just using the network (and taking connectivity as granted).

Most of the challenges here are related to compute resource allocation, i.e., orchestration methods for instantiating the right compute instance on a corresponding platform -- for achieving fault tolerance, performance optimization and cost reduction.

Examples of successful applications of networked computing are typical overlay systems such as CDNs. As overlays they do not need to be "in the network" -- they are effectively applications. (Note: we sometimes refer to CDN as an "in-network" service because of the mental model of HTTP requests that are being directed and potentially forwarded by CDN systems. However, none of this happens "in the network" -- it is just a successful application of HTTP and underlying transport protocols.)

3.2. Packet Processing

Packet processing is a function "in the network" -- in a sense that middleboxes reside in the network as transparent functions that apply processing functions (inspection, classification, filtering, load management etc.) -- mostly _transparent_ to endpoints. Some middlebox functions (TCP split proxies, video optimizers) are more invasive in a sense that they do not only operate on IP flows but also try to impersonate transport endpoints (or interfere with their behavior).

Since these systems can have severe impacts on service availability, security/privacy, and performance they are typically not very _programmable_ -- they just execute (usually) static code for predefined functions.

Active Networking can be characterized as an attempt to offer abstractions for programmable packet processing from an "endpoint perspective", i.e., by using data packets to specify intended behavior in the network with the aforementioned security problems.

Programmable Data Plane approaches such as P4 are providing abstractions of different types of network switch hardware (NPUs, CPUs, FPGA, PISA) from a switch/network programming perspective. The corresponding programs are constrained by the capabilities (instruction set, memory) of the target platform and typically operate on packets/flow abstractions (for example match-action-style processing).

Network Functions Virtualization (NFV) is essentially a "Networked Computing" approach (after all, Network Functions are just virtualized compute functions that get instantiated on compute platforms by an orchestrator). However, some Virtual Network Functions (VNFs) happen to process/forward packets (e.g., gateways in provider networks, NATs or firewalls). Still, that does not affect their fundamental properties as virtualized computing functions.

When connecting VNFs, there is the question of how to steer packet flows so that packets reach the right functions (and pass through them in the right order). One way is through configuration and network control/management (SDN), i.e., the VNFs are places in a virtual network, and there are configurations for meaningful next-hop IP addresses etc.

A more dynamic way is through Service Function Chaining (SFC, [RFC7665]), where a dynamic chain of IP-addressable packet processors can be specified (in an encapsulation packet header structure) and where forwarding nodes are equipped to interpret these headers and forward the packets to the appropriate next hops.

The SFC [RFC7665]) framework works with IP addresses for function (host) identifiers. Name-Based Service Function Forwarding [RFC8677] takes this one step further by adding another layer of indirection and by identifying the Service Functions using a name rather than a routable IP endpoint (or Layer 2 address). In addition to the naming concept, [RFC8677] also described the possibility of using different transport and application layer protocols for the communication between functions -- which could in principle extend the applicability from mere packet processing to some form of distributed computing.

3.3. Computing in the Network

In some deployments, networked computing and packet processing go well together, for example, when network virtualization (multiplexing physical infrastructure for multiple isolated subnetworks) is achieved through data-plane programming (SDN-style) to provide connectivity for VMs of a tenant system.

While such deployments are including both computing and networking, they are not really doing computing _in the network_. VM/containers are virtualized hosts/processes using the existing network, and packet processing/programmable networks is about packet-level manipulation. While it is possible to implement certain optimizations (for example, processing logic for data aggregation) -- the applicability is rather limited especially for applications where application-data units do not map to packets and where additional transport protocols and security requirements have to be considered.

Multi-access Edge Computing [MEC] is a particular architecture that leverages the virtual host platform concept, and that is focused on management and orchestration for such platforms. MEC can be combined with virtual networking concepts such as "Network Slicing" in 5G [MEC5G] to assure a certain QoS for connectivity to MEC platform instances. It should be noted that there may be other forms of edge computing that are not VM-based.

Distributed Computing (stream processing, edge computing) on the other side is an area where many application-layer frameworks exist that actually _could_ benefit from a better integration of computing and networking, i.e., from a new "computing in the network" approach.

For example, when running a distributed application that requires dynamic function/process instantiation, traditional frameworks typically deploy an orchestrator that keeps track of available host platforms and assigned functions/processes. The orchestrator typically has good visibility of the availability of and current load on host platforms, so it can pick suitable candidates for instantiating a new function.

However, it is typically agnostic of the network itself -- as application layer overlays the function instances and orchestrators take the network as a given, assuming full connectivity between all hosts and functions. While some optimizations may still be feasible (for example co-locating interacting functions/processes on a single host platform), these systems cannot easily reason about

* shortest paths between function instances; function off-loading

* opportunities on topologically convenient next hops; and

* availability of new, not yet utilized resources in the network.

While it is possible to perform optimizations like these in application layers overlays, it involves significant monitoring effort and would often duplicate information (topology, latency) that is readily available inside the network. In addition to the

associated overhead, such systems also operate at different time scales so that direct reaction in fine-grained computing environments is difficult to achieve.

When asking the question of how the network can support distributed computing better, it may be helpful to characterize this problem as a resource allocation optimization problem: Can we integrate computing and networking in a way that enables a joint optimization of computing and networking resource usage? Can we apply this approach to achieve certain optimization goals such as:

* low latency for certain function calls or compute threads;

* high throughput for a pipeline of data processing functions;

* high availability for an overall application/service;

* load management (balancing, concentration) according to performance/cost constraints; and

* consideration of security/privacy constraints with respect to platform selection and function execution?

* Also: can we do this at the speed of network dynamics, which may be substantially higher than the rate at which distributed computing applications change?

Considering computing and networking resource holistically could be the key for achieving these optimization goals (without considerable overhead through telemetry, management and orchestration systems). If we are able to dissolve the layer boundaries between the networking domain (that is typically concerned with routing, forwarding, packet/flow-level load balancing) and the distributed computing domain (that is typically concerned with 'processor' allocation, scaling, reaction to failure for functions/processes), we might get a handle to achieve a joint resource optimization and enable the distributed computing layer to leverage network-provided mechanisms directly.

For example, if distributing information about available/suitable compute platform could be a routing function, we might be able to obtain and utilize this information in a distributed fashion. If instantiating a new function (or offloading some piece of computation) could consider live performance data obtained from a in-network forwarding/offloading service (similar to IP packet forwarding in traditional IP networks), the "next-hop" decision could be based both on network performance and node load/availability).

Integrating computing and networking in this manner would not rule out highly optimized systems leveraging sophisticated orchestrators. Instead, it would provide a (possibly somewhat uniform) framework that could allow several operating and optimization modes, including totally distributed modes, centralized orchestration, or hybrid forms, where policies or intents are injected into the distributed decision-making layer, i.e., as parameters for resource allocation and forwarding decisions.

3.4. Elements for Computing in the Network

In-network computing requires computing resources (CPU, possibly GPUs, memory, ...), physical or virtualized to some extent by a suitable platform. These computing resources may be available in a number of places, as partly already discussed above, including the following:

* They may be found on dedicated machines co-locating with the routing infrastructure, e.g., having a set of servers next to each router as one may find in access network concentrators. This would come closest to today's principles of edge computing.

* They may be integrated with routers or other network operations infrastructure and thus be tightly integrated within the same physical device.

* They may be integrated within switches, similar to the (limited) P4 compute capabilities offered today.

* They may be located on NICs (in hosts) or line cards (routers) and be able to proactively perform some application functions, in the sense of a generalized variant of "offloading" that protocol stacks perform to reduce main CPU load.

* They might add novel types of dedicated hardware to execute certain functions more efficiently, e.g., GPU nodes for (distributed) analytics.

* They might include low-end (embedded) devices such as microcontrollers that support decentralized computation at low cost and limited performance.

* They may also encompass additional resources at the edge of the network, such as sensor nodes. Associated sensors could be physical (as in IoT) or logical (as in MIB data about a network device).

* Even user devices along the lines of crowd computing [CROWD] or mist computing [MIST] may contribute compute resources and dynamically become part of the network.

Depending on the type of execution platform, as already alluded to above, a suitable execution framework must be put in place: from lambda functions to threads to processes or process VMs to unikernels to containers to full-blown VMs. This should support mutual isolation and, depending on the service in question, a set of security features (e.g., authentication, trustworthy execution, accountability). Further, it may be desirable to be able to compose the executable units, e.g., by chaining lambda functions or allowing unikernels to provide services to each other -- both within a local execution platform and between remote platform instances across the network.

The code to be executed may be pre-installed (as firmware, as microcode, as operating system functions, as libraries, as *aaS offering, among others) or may be dynamically supplied. While the former is governed by the entity operating the execution device or supplying it (the vendor), the code to be executed may have different origins. Fundamentally, we can distinguish between two cases:

1. The code may be "centrally" provisioned, originating from an application or other service provider inside the network. This is analogous to CDNs, in which an application provider contracts a CDN provider to host content and service logic on its behalf. The deployment is usually long-term, even if instantiations of the code may vary. The code thus originates from rather few -- known -- sources. In this setting, applications only invoke this code and pass on their parameters, context, data, etc.

2. The code may be provided in a decentralized manner from a user device or other service that requires a certain function or service to be carried out. At the coarse granularity of entire application images, this has been explored as "code offloading"; recent approaches have moved towards finer granularities of offloading (sets of) functions, for which also some frameworks for smartphones were developed, leading to finer granularities down to individual functions. In this setting, application transfer mobile code -- along with suitable parameters, etc. -- into the network that is executed by suitable execution platforms. This code is naturally expected to be less trusted as it may come from an arbitrary source.

Obviously, 1. and 2. may be combined as mobile code may make use of other in-network functions and services, allowing for flexible application decomposition. Essentially, computing in the network may

support everything from full application offloading to decomposing an application into small snippets of code (e.g., at class, objects, or function granularity) that are fully distributed inside the network and executed in a distributed fashion according to the control flow of the application. This may lead to iterative or recursive calling from application code on the initiating host to mobile code to pre-provisioned code.

Another dimension beyond where the code comes from is how tightly the code and the data are coupled. At one extreme, approaches like Active Messages combine the data and the code that operates (only) on that data into transmission units, while at the other extreme approaches like Network Function Virtualization are only concerned with the instantiation of the code in the network. The underlying architectural question is whether the goal is to enable the network to perform computations on the data passing through it, or whether the goal is to enable distributed computational processes to be built in the network. And, of course, complete applications may leverage both approaches.

With these different existing and possibly emerging platforms and execution environments and different ways to provision functions in the network, it does not seem useful to assume any particular platform and any particular "mobile code" representation as _the_ "computing in the network" environment. Instead, it seems more promising to reason about properties that are relevant with respect to distributed program semantics and protocols/interfaces that would be used to integrate functions on heterogeneous platforms into one application context. We discuss these ideas and associated challenges in the following section.

4. Examples

4.1. Compute-First Networking with ICN

[CFN] is an example of a computing-in-the-network system that is based on computation graph representation for distributed programs. These programs are composed of stateful actors and stateful functions that are dynamically instantiated on available compute resources.

The first motivating use case was a real-time health monitoring system that analyzed audio samples from coughing noises which involves processing several audio feeds for noise addition and subtraction and for feature extraction.

The key concept of CFN is to provide a general-purpose distributed computing framework that can be programmed without knowledge about the runtime environment but that can leverage the dynamic resource properties automatically, and with reasonable efficiency.

CFN can lay out compute graphs over the available computing platforms in a network to perform flexible load management and performance optimizations, taking into account function/actor location and data location, as well as platform load and network performance.

In CFN, compute nodes that can execute functions within a given program instance are called workers. The allocation of functions and actors to workers happens in a distributed fashion. A CFN system knows the current utilization of available resources and the least cost paths to copies of needed input data. It can dynamically decide which worker to use, performing optimizations such as instantiating functions close to big data inputs. The bindings that control which execution platforms host which program interfaces (or individual functions/actors) is maintained through a computation graph.

To realize this distributed scheduling, workers in each resource pool advertise their available resources. This information is shared among all workers in the pool. A worker execution environment can decide, without a centralized scheduler, which set of workers to prefer to invoke a function or to instantiate an actor. In order to direct function invocation requests to specific worker nodes, CFN utilizes the underlying ICN network's forwarding capabilities -- the network performs late binding through name-based forwarding and workers can provide forwarding hints to steer the flow of work.

4.2. Akka Toolkit

The Akka toolkit (https://akka.io/) for building concurrent and distributed applications on the the JVM that is used by frameworks such as Apache Flink (https://flink.apache.org/). Akka is implements the Actor model, a way of realizing distributed computing as asynchronous message-based communication between concurrent processes that encapsulate application logic.

Communication between distributed actors is based on symmetric peer-to-peer model (actors can send each other messages) and is implemented by TCP-based protocols (https://doc.akka.io/docs/akka/2.3/scala/remoting.html).

Akka actors are logically organized in a tree hierarchy (https://doc.akka.io/docs/akka/current/general/addressing.html), and there are two addressing concepts: 1) Actor References that unique identify an actor instance and 2) Actor Paths, hierarchically

structured names that specify the logical position of an actor instance in system tree. Actor path can have an address component that specified location information (e.g., host and port number).

Akka has a routing concept (https://doc.akka.io/docs/akka/current/typed/routers.html) that can duplicate and distribute messages to a set of actors (for example for map-reduce like parallelism).

The Akka toolkit support cluster features (https://doc.akka.io/docs/akka/current/typed/cluster.html), i.e., the management of a collection of JVMs that can be monitored for resource and failure management.

4.3. Distributed Stream Processing

Stream Processing typically refers to systems that can query and process continuous streams of data, for example for data analytics. Such systems are often composed of individual functions that are arranged in a graph in which one function consumes output from an upstream function. These functions can be distributed and run concurrently, for example on multiple CPUs or multiple nodes in a network.

In typical systems such as Apache Storm [STORM] and Apache Flink [FLINK], the stream processing application logic can be specified as a graph in a high-level specification language, and then a corresponding framework is responsible for translating the graph into an operational run-time system and executing it. For example, this involves instantiating functions on available compute nodes and establishing some form on connectivity between the functions.

At run-time, some systems can be scaled out, i.e., depending on offered load and observed performance, some elements of the compute graph can be duplicated and then run in parallel. Other run-time operations can include failure management, i.e., re-starting or replacing failed components, and optimizations such as re-locating, e.g., consolidating functions onto compute nodes.

4.4. Distributed Machine Learning

Distributed Machine Learning [DML] refers to dividing large training jobs across multiple processors while training large deep learning models. These systems can be classified into two top-level categories: systems with 1) model parallelism and 2) data parallelism.

Model parallelism refers to systems where the model is split into partitions that are processed by different compute nodes and where these nodes communicate for training and for performing inference. Two sub-variants can be distinguished: vertical splitting (between the neural network layers) and horizontal splitting (between the individual layers). Vertical splitting is easier and more common.

Data parallelism refers to systems where the model is replicated onto several nodes, and where each node performs its own backpropagation in parallel to other nodes. The respective results are aggregated and integrated into a new model (typically continuously). When splitting the input data between different models, parallel training and thus performance gains can be achieved. This approach is also referred to as Federated Learning.

The two training steps, gradient computation and optimization, can be arranged in different ways: in centralized optimization, there is a central server for executing the optimization step whereas the gradient computation happens on a set of worker nodes. In decentralized optimization, both steps are replicated in each worker.

Distributed training can use either synchronous or asynchronous scheduling, enforcing a loser or tighter coupling between workers.

Communication and computation performance can obviously affect the overall distributed training performance significantly, and depending on the specific variant, distributed learning systems require certain coordination between workers (and servers).

Federated learning systems typically employ a simple topology of parallel workers and a centralized server.

Challenges in this field include mapping the distributed learning optimally to a given infrastructure, splitting the data accordingly, achieving effective and efficient synchronization and coordination, and dealing with dynamically changing network characteristics, e.g., when running over a shared, potentially unreliable infrastructure.

5. Research Challenges

Let us take the above notion of computing in the network as a joint resource optimization problem as a starting point. This joint resource management in itself is already a notable research challenge, especially when tackled for operation at network time scales and Internet-wide user application scales. But there are further research challenges from perspectives of modeling and abstractions, systems consideration, protocol aspects, and application design paradigms, among others. We will discuss (an

admittedly unlikely complete set of) these in this section.

5.1. Categorization of Different Use Cases for Computing in the Network

There are different applications but also different configuration classes of Computing in the Network systems. For example, a data processing pipeline might be different from a distributed application employing some stateful actor components. It is worthwhile analyzing different typical use cases and identify commonalities (for example, fundamental protocol elements etc.) and differences. It is equally important to critically assess which of these use cases truly belong to the class of Computing in the Network as opposed to networked or edge computing, acknowledging that the boundaries may be fluent.

An ongoing effort to this end is elaborated on in a companion document [I-D.irtf-coinrg-use-cases].

5.2. Modeling Distributed Computing

Distributed systems can be modeled with several architecture patterns, e.g., client-server, peer-to-peer, and directed Acyclic Graphs (DAGs) as in some stream processing systems. A particular distributed application, e.g., a stream processing graph can be formally specified, i.e., by listing the involved functions and their relationship with respect to data processing. In other systems, callgraph structures are implicitly derived from a computer program.

In principle, it is possible to reason about computational complexity and resource requirements, i.e., with respect to computation, communication and storage resources, however general-purpose systems with Turing complete computation components make this difficult. Hence, such reasoning is often fairly coarse-grained (e.g., specify the class of computer server that a certain function needs) and/or based on heuristics or blackbox observations.

In general, the potential for modeling computing depends greatly on the structure of the distributed system and on the nature of the individual functions. DAG-based systems with simpler, more homogeneous compute functions, such as a deep learning layer behave more predictively and allow for some degree of modeling with respect to required resources.

5.3. Mapping Computing Semantics to Infrastructure

For instantiating and operating a distributed computing system in the network, the system's application logic, i.e., the semantic operations, need to be mapped to specific infrastructure, e.g., a network of compute nodes. Ideally, such a step would take the model of the specific application into consideration (e.g., computing complexity and other resource required) and then derive a suitable, ideally optimal mapping to the available infrastructure.

This could either happen statically, i.e., initially when allocating resources for a certain application, or dynamically and repeatedly, i.e., at run-time, taking changing application requirements, such as varying input data rates, and the current resource utilization situation into account. Especially when running over shared infrastructure, such changes are generally hard to predict, so challenges include a correct assessment of the resource optimization and adaptation algorithms that work well in the presence of multiple competing workloads.

Such resource management is often referred to as orchestration (which we discuss further in Section 5.8).

5.4. Networking and Remote-Method-Invocation Abstractions

In distributed systems, there are different classes of functions that can be distinguished, for example:

1. Strictly stateless functions that do not keep any context state beyond their activation time

2. Stateful functions/modules/programs that can be instantiated, invoked and eventually destroyed that do keep state over a series of function invocations

Modern frameworks such as Ray are offering a clear separation of stateless functions and stateful actors and offer corresponding abstractions in their programming environment. The aforementioned analysis of use cases should provide a diverse set of use cases for deriving a minimal yet sufficient set of function classes.

Beyond this fundamental categorization of functions/actors, there is the question of interfaces and protocols mechanisms -- as building blocks to utilize functions in programs. For example, stateful functions are typically invoked through some Remote Method Invocation (RMI) protocol that identifies functions, allows for specifying/ transferring parameters and function results etc. Stateful actors could provide class-like interfaces that offer a set of functions (some of which might manipulate actor state).

Another aspect is about identity (and naming) of functions and actors. For actors that are typically used to achieve real-world effects or to enable multiple invocations of functions manipulating actor state over time, it is obvious that there needs to be a concept of specific instances. Invoking an actor function would then require specifying some actor instance identifier.

Stateless functions may be different: an invoking instance may be oblivious with respect to the specific function instance and locus (on an execution platform) and might just want to leave it to the network to find the "best" instance or locus for a new instantiation. Some fine-granular functions might just be instantiated for one invocation. On the other hand, a function might be tied to a particular execution platform, for example an GPU-supported host system. The naming and identity framework must allow for specifying such a function (or at least equivalence classes) accordingly.

Stateful functions may share state within the same program context, i.e., across multiple invocations by the same application (as, e.g., holds for web services that preserve context -- locally or on the client side). But stateful functions may also hold state across applications and possibly across different instantiations of a function on different compute nodes. Such will require data synchronization mechanisms and the implementation of suitable data structure to achieve a certain degree of consistency. The targeted degree of consistency may vary depending on the function and so may the mechanisms used to achieve the desired consistency.

In cloud practice, serverless functions are usually stateless but they may achieve their stateless operation by pushing application state to an external storage entity, e.g., a key-value store, with the implicit assumption that any function instance would have equally fast access to the stored state. While researchers have explored offering similar storage capabilities for edge computing, this simplification may not hold for the edge and even less so for computing in the network. Hence, in-network functions and programs may need to consciously perform their state management.

Finally, execution platforms will require efficient resource management techniques to operate with different types of stateless and stateful functions and their associated resources, as well as for dynamically instantiated mobile code. Besides the aforementioned location of suitable compute platforms and scheduling (possibly queuing) functions and function invocations, this also includes resource recovery ("garbage collection").

5.5. Transport Abstractions

When implementing Computing in the Network and building blocks such as function invocation it seems that IP packet processing is not the right abstraction. First of all, carrying the context for some function invocation might require many IP packets -- possibly something like Application Data Units (ADUs). But even if such ADUs could be fit into network layer packets, other problems still need to be addressed, for example message formats, reliability mechanisms, flow and congestion control etc.

It could be argued that today's distributed computing overlays solve that by using TCP and corresponding application layer formats (such as HTTP) -- however this begs the question whether a fine-granular distributed computing system, aiming to leverage the network for certain tasks, is best served by a TCP/IP-based approach that entails issues such as

* need for additional resolution/mapping system to find IP addresses for functions;

* possible overhead for establishing TCP connections for fine-granular function invocation;

* defining and managing security properties of such connections and coping with the associated setup/validation overhead; and

* mismatch between TCP end-to-end semantics and the intention to defer next-hop selection etc. to the network.

Moreover, some Computing in the Network applications such as Big Data processing (Hadoop-style etc.) can benefit significantly from data-oriented concepts such as

* in-network caching (of data objects that represent function parameters or results);

* reasoning about the tradeoffs between moving data to function vs. moving code to data assets; and

* sharing data (e.g., function results) between sets of consuming entities.

RMI systems such as RICE [RICE] enable Remote Method Invocation of ICN (data-oriented network/transport). Research questions include investigating how such approaches can be used to design general-purpose distributed computing systems. More specifically, this would involve questions such as:

* What is the role of network elements in forwarding RMI requests?

* What visibility into load, performance and other properties should endpoints and the network have to make forwarding/offloading decisions and how can such visibility be afforded? What are and how to control the security implications of such visibility?

* What is the notion of transport services in this concept and how intertwined is traditional transport with RMI invocation?

* What kind of feedback mechanisms would be desirable for supporting corresponding transport services?

Moreover, it is to be noted that RMI flavors are unlikely suitable, or at least: efficient, for all kinds of function interactions. For example, real-time data flows and stream processing would likely benefit from other abstractions. Identifying the needs, classifying them into abstraction categories, and devising feasible transport abstractions and mapping them to (existing or new developed/adapted) protocols constitute further research challenges, for which the aforementioned questions apply equally.

5.6. Programming Abstractions

When creating SDKs and programming environments (as opposed to individual point solutions) questions arise such as:

* How to use concepts such as stateless functions, actor models and RMI in actual programs, i.e., what are minimal/ideal bindings or extensions to programming languages so that programmers can take advantage of Computing in the Network?

* Are there additional, potentially higher-layer, abstractions that are needed/useful, for example data set synchronization, data types for distributed computing such as CRDTs?

* How do these map meaningfully to the transport abstractions defined above?

In addition to programming languages, bindings, and data types, there is the question of execution environments and mobile code representation. With the vast number of different platforms (CPUs, GPUs, FPGAs etc.) it does not seem useful to assume exactly one environment. Instead, interesting applications might actually benefit from running one particular function on a highly optimized platform but are agnostic with respect to platforms for other, less performance-critical functions. Being able to support a heterogenous, evolving set of execution environments brings about questions such as:

* How to discover available platforms (and understand their properties)?

* How to specify application needs and map them to available platforms?

* Can a certain function/application service be provided with different fidelity levels, e.g., can an application leverage a GPU platform if available and fall back to a reduced feature set in case such a platform is not available?

* How to keep the complexity of these seemingly countless options under control so that, ultimately, efficient algorithms can be devised that can operate at the targeted (scale, timescale) pairs and interoperable systems can be built.

In this context, updates and versioning could entail another dimension of variability for Computing in the Network:

* How to manage coexistence of multiple versions of functions and services, also for service routing and request forwarding?

* Is there potential for fallback and version negotiation if needed (considering the risk of "bidding downs" attacks?)

* How to retire old versions?

* How to securely and reliably deal with function updates and corresponding maintenance tasks?

5.7. Security, Privacy, Trust Model

Computing in the Network has interesting security-related challenges, including:

* How can a caller trust that a remote function works as expected? This entails several questions such as

  - How to securely bind "function names" to actual function code?

  - How to trust the execution platform (in its entirety)?

  - How to trust the network that is forwards requests (and result messages) reliably and securely?

  - How to ascertain that a function does what it claims to do?

* What levels of authentication are needed for callers (assuming that not everybody can invoke any function)?

* How to authenticate and achieve confidentiality for requests, their parameters and result data (especially when considering sharing of results)?

Many of these questions are related to other design decisions such as

* What kind of session concept do we assume, i.e., is there a concept of distributed application session that represents a trust domain for its members?

* Where is trust anchored? Can the system enable decentralized operation?

All of these questions are not new, but conceiving networking and computing holistically seems to revisit distributed systems and network security -- because some established concepts and technologies may not be directly applicable (such as transport layer security and corresponding web PKI).

5.8. Orchestration and Coordination

For distributed systems, coordination is a key function and involves several functions such as configuration management, service discovery, application state management, and consensus schemes.

As noted above in Section 5.2 and Section 5.3, programs can be modeled as interactions of functions and then mapped to the available compute nodes in the infrastructure for execution. This very mapping process is often called orchestration and needs to take into account

* on the one hand, the application needs for computational power and possibly specific hardware as well as the interaction demands on network resources to connect the user to the functions and the functions to each other and,

* on the other hand, the available resources and possibly their current (and projected) utilization.

On orchestration function has then the task to find possible matches and choose one for a concrete allocation of the function instances. For this task, different search (and possibly optimization) algorithms can be devised, noting that there may be many utility functions for which these algorithms may optimize. Those are expected to differ for the user of a function (goal: best application performance), the provider of a program (goal: minimize cost), and the infrastructure provider or operator (goal: maximize utilization and profit).

Whatever the algorithm be, an initial placement and associated resource allocation will result. This may need to be revisited as time proceeds, new resource demands arise, users move, or programs complete, which leads of a repeated invocation of the respective algorithms. As a revised allocation may require stateful functions to be migrated (or stateless ones to be terminated and newly instantiated), the cost of migrating functions also needs to be accounted for in the decision making. There are many opportunities to explore for novel resource management algorithms and utility functions and their efficient implementation.

Programs, functions, and function instances require a naming architecture so that (repeated) function calls can be resolved to the (same) function instance as needed (cf. stateful functions). Once they are placed, the network plumbing (read: address resolution, routing, and/or forwarding) has to be configured to ensure that the respective functions can be found and reached (discovery) both by the user and by other functions. This gives rise to exploring various network designs, from simple dynamic DNS-based resolutions to anycasting to semantic addressing to overlay-based routing to information-centric and named-function networking and beyond.

How these functions are implemented depends a lot on the nature of specific systems. For example, Apache ZooKeeper (https://zookeeper.apache.org/) is a logically centralized coordination service that provides coordination primitives to client application modules. The ZooKeeper itself is implemented as a distributed system consisting of a set of tightly coupled server instances that replicate the ZooKeeper state.

Hierarchical variants such as Oakestra [OAKESTRA] support the decentralized management of resources by applying only a limited coupling between resources clusters and centralized managers and allowing multiple roots to oversee (possibly overlapping) pools of resources. This also supports federating resource clusters from different infrastructure providers.

Other systems, such as the ICN-based CFN Section 4.1 implement these services in a distributed way, employing different mechanisms for synchronization and consensus building.

While the fundamental concepts and mechanisms for coordination services are well understood, applying these concepts and mechanisms to a specific system design requires careful consideration.

5.9. Fault Tolerance, Failure Handling, Debugging, Management

Distributed computing naturally provides different types of failures and exceptions. In fine-granular distributed computing, some failures may by more tolerable (think microservices), i.e., platform crash or function abort due to isolated problems could be handled by just re-starting/re-running a particular function. Similarly, "message loss" or incorrect routing information may be repairable by the system itself (after time).

When failure cannot be repaired (or just tolerated) by the distributed computing framework, this raises questions such as:

* What are strategies for retrying vs aborting function invocation?

* How to signal exceptions and enable robust response to failures?

Failure handling and debugging also has a management aspect that leads to questions such as:

* What monitoring and instrumentation interfaces are needed?

* How can we represent, visualize, and understand the (dynamically changing) properties of Computing in the Network infrastructure as well as of the currently running/instantiated entities?

6. Acknowledgements

The authors would like to thank Dave Oran, Michal Krol, Spyridon Mastorakis, Yiannis Psaras, Eve Schooler, Dirk Trossen, and Phil Eardley for previous fruitful discussions on Computing in the Network topics and for feedback on this draft.

7. ChangeLog

7.1. 03

* new Section 4.3 on Distributed Stream Processing
* new Section 4.4 on Distributed Machine Learning
* new Section 5.2 on Modeling Distributed Computing
* new Section 5.3 on Mapping Computing Semantics to Infrastructure
* new text on orchestration in Section 5.8
* misc. additions throughout

7.2. 02

* fixed errors and updates references
* new Section 5.8 on Coordination
* renamed Section 5.9 to Fault Tolerance, Failure Handling, Debugging, Management
* new Section 4.2 on Akka in Section 4

7.3. 01

* added explanation of MEC and network slicing in Section 3.
* added clarification that edge computing is not limited to MEC
* added description of named service function chaining
* new Section 4 with a description of CFN-ICN

8. Informative References

[ACTIVE] Tennenhouse, D. L., Wetherall, D. J., and Association for Computing Machinery (ACM), "Towards an active network architecture", ACM SIGCOMM Computer Communication Review, vol. 26, no. 2, pp. 5-17, DOI 10.1145/231699.231701, 15 April 1996, <https://doi.org/10.1145/231699.231701>.

[CANARY] Qu et al, H., "Canary -- A scheduling architecture for high performance cloud computing", 2016, <https://arxiv.org/abs/1602.01412>.

[CFN] Król, M., Mastorakis, S., Oran, D., Kutscher, D., and ACM, "Compute First Networking", Proceedings of the 6th ACM Conference on Information-Centric Networking, DOI 10.1145/3357150.3357395, 24 September 2019, <https://doi.org/10.1145/3357150.3357395>.

[CROWD] Murray, D. G., Yoneki, E., Crowcroft, J., Hand, S., and ACM, "The case for crowd computing", Proceedings of the second ACM SIGCOMM workshop on Networking, systems, and applications on mobile handhelds, DOI 10.1145/1851322.1851334, 30 August 2010, <https://doi.org/10.1145/1851322.1851334>.

[DML] Langer, M., He, Z., Rahayu, W., Xue, Y., and Institute of Electrical and Electronics Engineers (IEEE), "Distributed Training of Deep Learning Models: A Taxonomic Perspective", IEEE Transactions on Parallel and Distributed Systems, vol. 31, no. 12, pp. 2802-2818, DOI 10.1109/tpds.2020.3003307, 1 December 2020, <https://doi.org/10.1109/tpds.2020.3003307>.

[EDGESURVEY] Mach et al, P., "Mobile Edge Computing -- A Survey on Architecture and Computation Offloading", 2017, <https://ieeexplore.ieee.org/document/7879258>.

[FLINK] Katsifodimos, A., Schelter, S., and IEEE, "Apache Flink: Stream Analytics at Scale", 2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW), DOI 10.1109/ic2ew.2016.56, April 2016, <https://doi.org/10.1109/ic2ew.2016.56>.

[FOGEDGE] Salaht, F. A., Desprez, F., Lebre, A., and Association for Computing Machinery (ACM), "An Overview of Service Placement Problem in Fog and Edge Computing", ACM Computing Surveys, vol. 53, no. 3, pp. 1-35, DOI 10.1145/3391196, 12 June 2020, <https://doi.org/10.1145/3391196>.

[I-D.irtf-coinrg-use-cases] Kunze, I., Wehrle, K., Trossen, D., Montpetit, M., de Foy, X., Griffin, D., and M. Rio, "Use Cases for In-Network Computing", Work in Progress, Internet-Draft, draft-irtf-coinrg-use-cases-04, 30 June 2023, <https://datatracker.ietf.org/doc/html/draft-irtf-coinrg-use-cases-04>.

[MEC] ETSI, "Multi-access Edge Computing (MEC)", 2020, <https://www.etsi.org/technologies/multi-access-edge-computing>.

[MEC5G] Sami Kekki et al, "MEC in 5G Networks", 2018, <https://www.etsi.org/images/files/ETSIWhitePapers/etsi_wp28_mec_in_5G_FINAL.pdf>.

[MIST] Barik, R. K., Dubey, A. C., Tripathi, A., Pratik, T., Sasane, S., Lenka, R. K., Dubey, H., Mankodiya, K., Kumar, V., and Elsevier BV, "Mist Data: Leveraging Mist Computing for Secure and Scalable Architecture for Smart and Connected Health", Procedia Computer Science, vol. 125, pp. 647-653, DOI 10.1016/j.procs.2017.12.083, 2018, <https://doi.org/10.1016/j.procs.2017.12.083>.

[OAKESTRA] Bartolomeo, G., Bäurle, S., Mohan, N., Ott, J., and ACM, "Oakestra", Proceedings of the SIGCOMM '22 Poster and Demo Sessions, DOI 10.1145/3546037.3546056, 22 August 2022, <https://doi.org/10.1145/3546037.3546056>.

[RAY] Moritz et al, P., "Ray -- A Distributed Framework for Emerging AI Applications", 2018, <http://dl.acm.org/citation.cfm?id=3291168.3291210>.

[RFC7665] Halpern, J., Ed. and C. Pignataro, Ed., "Service Function Chaining (SFC) Architecture", RFC 7665, DOI 10.17487/RFC7665, October 2015, <https://www.rfc-editor.org/info/rfc7665>.

[RFC8677] Trossen, D., Purkayastha, D., and A. Rahman, "Name-Based Service Function Forwarder (nSFF) Component within a Service Function Chaining (SFC) Framework", RFC 8677, DOI 10.17487/RFC8677, November 2019, <https://www.rfc-editor.org/info/rfc8677>.

[RICE] Król, M., Habak, K., Oran, D., Kutscher, D., Psaras, I., and ACM, "RICE", Proceedings of the 5th ACM Conference on Information-Centric Networking, DOI 10.1145/3267955.3267956, 21 September 2018, <https://doi.org/10.1145/3267955.3267956>.

[SAPIO] Sapio, A., Abdelaziz, I., Aldilaijan, A., Canini, M., Kalnis, P., and ACM, "In-Network Computation is a Dumb Idea Whose Time Has Come", Proceedings of the 16th ACM Workshop on Hot Topics in Networks, DOI 10.1145/3152434.3152461, 30 November 2017, <https://doi.org/10.1145/3152434.3152461>.

[SPARROW] Ousterhout, K., Wendell, P., Zaharia, M., Stoica, I., and ACM, "Sparrow", Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, DOI 10.1145/2517349.2522716, 3 November 2013, <https://doi.org/10.1145/2517349.2522716>.

[STORM] Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J. M., Ramasamy, K., Taneja, S., and ACM, "Twitter Heron", Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, DOI 10.1145/2723372.2742788, 27 May 2015, <https://doi.org/10.1145/2723372.2742788>.

Authors' Addresses

Dirk Kutscher
HKUST(GZ)
No 1 Du Xue Road, Nansha District
Guangzhou
China

Email: dku@ust.hk

Teemu Kaerkkaeinen
Technical University Muenchen
Boltzmannstrasse 3
Munich
Germany

Email: kaerkkae@in.tum.de

Joerg Ott
Technical University Muenchen
Boltzmannstrasse 3
Munich
Germany

Email: jo@in.tum.de