Position Paper

A case for object capabilities as the foundation of a distributed environmental model and simulation infrastructure

Michael Berg-Mohnicke\textsuperscript{a,}\textsuperscript{*}, Claas Nendel\textsuperscript{a,b,c}

\textsuperscript{a} Leibniz Centre for Agricultural Landscape Research (ZALF), Eberswalder-Str. 84, 15374, Müncheberg, Germany
\textsuperscript{b} Institute of Biochemistry and Biology, University of Potsdam, Am Mühlenberg 3, 14476, Potsdam (Golm), Germany
\textsuperscript{c} Global Change Research Institute, The Czech Academy of Science, Bělidla 986/4a, 603 00, Brno, Czech Republic

\textbf{ARTICLE INFO}

\textbf{Keywords:}
- Cap' n' proto
- Scientific collaboration
- Co-development
- Communication protocol
- Object capability

\textbf{A B S T R A C T}

With the advent of increasingly powerful computational architectures, scientists use these possibilities to create simulations of ever-increasing size and complexity. Large-scale simulations of environmental systems require huge amounts of resources. Managing these in an operational way becomes increasingly complex and difficult to handle for individual scientists. State-of-the-art simulation infrastructures usually provide the necessary resources in a centralised setup, which often results in an all-or-nothing choice for the user. Here, we outline an alternative approach to handling this complexity, while rendering the use of high-performance hardware and large datasets still possible. It retains a number of desirable properties: (i) a decentralised structure, (ii) easy sharing of resources to promote collaboration and (iii) secure access to everything, including natural delegation of authority across levels and system boundaries. We show that the object capability paradigm will cover these issues, and present the first steps towards developing a simulation infrastructure based on these principles.

1. Introduction

1.1. The environmental modelling context

In a rapidly changing world, science is increasingly called on to project future developments of our societal systems and the environmental systems in which they are embedded (Hofman et al., 2017; MEA, 2005; Meadows et al., 1972). Story-telling and the development of narratives is a commonly agreed way of outlining desirable or undesirable futures in a qualitative manner, while simulations are used to translate the rather fuzzy narratives into a quantitative universe (Malampalli et al., 2016; Schönenberg et al., 2017). There are already numerous feedback loops within environmental and societal systems, and interactions between the two systems intensify the complexity. A vast number of simulation models have been developed to represent various sub-systems at different levels of detail. Their purposes range from enhancing the understanding of the system to predicting the behaviour of the system when surrounding conditions change (Hofman et al., 2017; van Nes and Scheffer, 2005). Simulation models for environmental and societal systems include a range of methodological approaches and disciplinary foci, addressing questions of public and scientific concern (Ihrig, 2016; Verburg et al., 2016). The aim of integrated modelling is to combine individual modelling approaches into larger model frameworks and to utilise the feedback loops across the different models to gain a better understanding and predictability of the systems represented (van Beek et al., 2020; van Ittersum et al., 2008). However, the operation of individual models often requires a certain infrastructure (computing facilities, software environment, access to data, etc.; Verweij et al., 2010) and experience on the part of the user to produce the desired output (Confalonieri et al., 2016), which is often limited to individual groups of developers.

Semantic and sociological challenges remain, but software, hardware and data have undergone substantial improvements in recent decades. These improvements enable increasingly complex simulations and new modelling approaches, also across domains, partially driving the development of even more powerful computers. As the potential for increasing the speed of single processors has hit the ceiling for now, parallel computing on multiple processors is on the rise (Shalf, 2020). At the same time, the advancement of sensors, sensor platforms and other forms of continuous and large-scale data creation has boosted the availability of big data for potential use in developing, calibrating or driving simulation models (Basso and Antle, 2020; Franz et al., 2020; Hampf et al., 2021; Weiss et al., 2020). In many cases, large institutionalised infrastructures are needed to harvest, store and serve such...
data. However, high-performance computing (HPC) facilities, big data repositories and skilled model developers are unevenly distributed across research institutions. The integration of different models and the utilisation of computing power and data availability therefore require cross-institutional collaboration and a software infrastructure that supports such collaboration (Altschul et al., 2013). Ideally, this infrastructure should integrate models, data and people, including model developers and model users. Drawing on the example of impact modelling in agriculture, we present below the vision and rationale for a flexible cooperative simulation infrastructure and the underlying software mechanisms needed to support it.

1.2. Criteria for a simulation infrastructure

No matter how it is done, designing and developing a new model and simulation infrastructure is an extensive undertaking. In light of past experiences and projects in the environmental modelling field, an ideal infrastructure has the following desirable properties:

1 **Flexibility**: The ideal infrastructure allows for easy integration of diverse hardware (PC, HPC, Cloud) and software (legacy, different languages, etc.) and can handle a wide range of heterogeneous models and data.

2 **Efficiency**: The ideal infrastructure facilitates the fast and efficient execution of both simple and complex simulations, and does not represent the bottleneck of the simulation.

3 **Simplicity**: The simpler the infrastructure, the higher the level of adoption. An infrastructure must be easy for scientists to use and for developers to improve.

4 **Peripherality**: To ease collaboration, the infrastructure should support the distributed and decentralised use of models and data.

5 **Openness**: A modern simulation infrastructure should be open to improvement by both scientists and developers, involving the collaboration of many different people. Ideally, such collaboration is voluntary, unconditional, easy and secure.

Today’s model and simulation infrastructures tend to be based on a centralised approach where a large storage and computational backend is accompanied by some user-facing, usually web-based, frontend that allows access to the system. Consequently, models and data to be used within such an infrastructure must be prepared accordingly to fit into the system. The more commonality can be enforced, the more benefits can be gained by the easy use of models for new data or new models for existing data. What these centralised infrastructures have in common is that they act as silos, allowing only the models and data placed inside them to interact or utilise future enhancements of the system. This means that existing models and data that are to be included in the infrastructure have to be adapted to the infrastructure and its prevailing rules. Depending on the properties of the system, adjusting to the infrastructure requires a considerable amount of resources and skills. While the trade-off between costs and benefits can be very positive for large organisations, major projects or long-term needs, it may discourage smaller stakeholders, hamper agile experimentation and generally cause problems when models and/or data do not fit easily into the concept of an infrastructure. Similarly, it is difficult and expensive to cater for multiple, possibly competing infrastructures. In addition, centralised infrastructures encourage a centralised user and rights management system, which makes administrative control easy, but external collaboration and participation more difficult.

In order to mitigate some of the aforementioned problems and to achieve the properties required by the scientific community, we propose a different take on simulation infrastructure. Our goal is to create a potentially large, ad-hoc decentralised simulation infrastructure comprising networked components, such as computing or storage resources. All components are treated as equal members of the larger virtual infrastructure. These distributed components communicate via a capability-based remote procedure protocol (RPC) according to capability security principles. This allows fine-grained decentralised security and the easy and safe delegation of authority. It removes the need for centralised user management, enabling unconditional collaboration at all scales. Since all parts of the infrastructure are treated as equal, there is no distinction between simple and complex models, personal laptops and HPCs, or small local and large cloud-backed datasets. What counts at the infrastructure level is communication between the components that make up a simulation. It goes without saying that different needs will result in the creation of different setups. An organisation’s need for high-performance, large-scale simulations will result in parts of the virtual infrastructure – models and data – being optimised for HPCs. On the other hand, a scientist might want to hook into the same setup, and prototype a model change or experiment using a new, locally crafted dataset. In both cases, users need access to the components representing the resources in order to run a particular simulation. The authority to access such resources can be delegated by peers or granted by administrators, as is the case in centralised systems. Decentralisation does not mean the complete abandonment of centralisation. Instead, it seeks to establish multiple centres, and potentially lots of them. The costs of the virtual infrastructure are then reduced to the cost of self-hosted systems and a negotiable part for remotely used systems. It is likely that only simulations bound by performance needs will have to run in a purely centralised manner. The simulation infrastructure we propose attempts to bridge the extremes (local vs HPC) by offering sufficient flexibility to fill the space between those extremes.

1.3. The transition from centralised to decentralised systems

Decentralisation requires that all parts communicate. Since speed and latency are always important, an efficient communication protocol, a common language and shared communication interfaces – both explicitly and externally specified – are needed. Agreeing on a common set of interfaces for the infrastructure’s components allows to abstract away some of the heterogeneity of real-world systems, and renders possible the creation of a virtual infrastructure. Stable communication interface specifications encourage model creators and data suppliers to implement them for their own resources. Equipped with sufficient tooling, end users may also become model or data providers, being able to hook into the infrastructure peripherally.

Security is an even more important issue for collaboration in open distributed systems than it is in closed centralised infrastructures. While the latter can employ well-established user and rights management mechanisms, controlled by an administrator with elevated rights, a single point of control is difficult and ultimately not desirable in a decentralised system. To allow as much freedom as possible within the infrastructure, we selected the capability security paradigm (Miller et al., 2003) as the underlying authority transfer and access mechanism. Capability security gives the user access to a particular resource and the authority to handle it (together, both will be defined later as “capability”) without a third party (e.g. an administrator) being involved, and the opportunity to pass on this capability to others. This transfer of capabilities is part of the communication protocol and, as such, a software entity that can address both users and machines. Capabilities can be transferred via messages that are otherwise used to execute a simulation model, for example, and pass on access information and rights. In the simulation model example, the model can receive access information to a particular data set plus the authority to access it.

All these factors are easily applicable as long as all the components are connected. However, clients may go offline, and servers and systems may fail or reboot. Users, once authorised, may want to retain the capability received between work sessions or pass them on to somebody else via infrastructure-independent means, such as an email. To this end, capabilities need to persist so that they can be stored offline and used at a later stage to gain access to the resources represented. Persistent capabilities are called sturdy references (erights.org, 1998a) or sturdy refs.
Receiving a sturdy ref enables a user or a software program to connect to the resource represented, and interact with that resource according to the capability’s communication interface.

1.4. Choice of a particular technology

So far, we have made only an abstract reference to the foundational capability-secure layer that we believe a decentralised simulation infrastructure needs to develop the desired properties. Since it is likely that a decentralised system will consist of very different hardware, software programs, programming languages, platforms and frameworks, the capability-secure communication protocol should be agnostic to languages and platforms. There is currently only one real-world implementation of a capability-secure communication protocol that meets our needs: being multi-language, multi-platform, offering an external schema description and focusing on speed and efficiency. Cap’n Proto (Cap’n Proto, 2022) is a fast data interchange format and a capability-based RPC protocol. It is an implementation of the CapTP protocol (CapTP, 2021), which allows “distributed object programming over mutually suspicious networks, meant to be combined with an object capability (ocap) style of programming” (Spiritely, 2021). In other words, it is a distributed object-oriented program secured by capability principles. Capabilities provided to scripts or programs act like a distributed object-oriented application programming interface (API), e.g. a script or program would start off with one or more sturdy refs (persistent capabilities) to some part of the infrastructure. Turning these into live references (e.g. to model instances or datasets) allows the program to access the currently available simulation API.

Now that the entry point to the simulation infrastructure is mediated by capabilities, software that uses and runs on top of the infrastructure can be secure even among mutually suspicious participants. A capability granted to a participant in the infrastructure should therefore only have the authority required for their tasks. Moreover, a capability should never allow unintended privilege escalation, which would involve an owner being granted more authority than is needed for a given task. This principle of least authority (POLA; Saltzer, 1974) can be expressed very naturally in object capability systems. The capability security principles are able to avoid many pitfalls of contemporary systems that are mainly based on ambient authority using access-control lists (ACLs), which we describe in more detail in the next section.

1.5. The example realm

Seeking to exemplify the concept of object capabilities in the context of a model and simulation infrastructure, we use the case of process-based agroecosystem modelling. Process-based agroecosystem models build on a system of differential equations that represent biophysical processes in agro-ecosystems to a larger region such as a country. The simulation model requires driving information in the form of time series that include climate data, soil data and management rules for each grid cell of the area to be covered.

2. Capability security and object capabilities

A capability is a communicable, unforgeable token of authority. An object capability, in turn, is an unforgeable reference to an object. The object’s interface defines the authority that can be exercised by sending messages to the object (Dennis and van Horn, 1966). In this section, we explain the concept of capability security and how it differs from the well-established mechanism of access control lists (ACLs), and address object capabilities in particular. The following examples are purposely reduced and simplified to allow focusing on the concepts. In the accompanying GitHub repository (Code, 2022) the complete source code, and the description how to run the examples, can be found.

2.1. Access control lists versus capabilities

Both access control lists and capabilities are used to control access to resources. Within a computer system, this requires a definition of which user (or process) has access to which resources. An ACL is a list of users and their access rights to a particular resource. Translated to a filesystem, a control authority (e.g. an administrator) specifies which users are allowed to access a file, and what are they allowed to do (e.g. read the file). In an ACL-based filesystem, a user refers to the resources (files) by the name of the file (the filename), which does not by itself entail the authority to access the file. Instead, authorisation is derived from the file’s environment. This concept is referred to as ambient authority. It may be viewed as an automatic door that identifies an authorised person on the basis of prior registration, and slides open when such a person approaches. In contrast, capabilities can be compared to keys. Equipped with the matching key, a person can open the door. In this case, access authorisation is attached to the key itself, and not to the environment.

A typical way of depicting the problem is to use a matrix showing the users along the vertical axis and the resources along the horizontal axis. The matrix in Fig. 2 states that all users are allowed to read the file at /etc/passwd, but only Alice and Carol are permitted to write to the file /u/markm/foo, whereas Bob is denied access to this file. ACLs and

<table>
<thead>
<tr>
<th>Resource</th>
<th>Alice</th>
<th>Bob</th>
<th>Carol</th>
</tr>
</thead>
<tbody>
<tr>
<td>/etc/passwd</td>
<td>{read}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>/u/markm/foo</td>
<td>{write}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>/etc/motd</td>
<td>{}</td>
<td>{}</td>
<td>{read}</td>
</tr>
</tbody>
</table>

Fig. 2. An access control matrix for three actors (Alice, Bob and Carol) and three resources (/etc/passwd, /u/markm/foo and /etc/motd) (Miller et al., 2003).
capabilities offer two different ways of looking at this matrix. In an ACL system, the file/etc/motd would have to have a list attached stating that both Bob and Carol are allowed to read the file. In contrast, in a capability system, Bob and Carol would need to have a key that allows them to read the file/etc/motd.

If the matrix is static, both approaches to defining access to a resource are equivalent. As soon as the matrix becomes dynamic, things differ. In the ACL approach, only the owner or privileged users (administrators) are usually allowed to change access lists. If, then, Alice also needs to read the file/etc/motd at a later point in time, the administrator has to update the ACL of the file/etc/motd. In a capability system, somebody with read access to the file/etc/motd has to give Alice (a copy of) the key so that she too can read the file. The important difference to note is that with capabilities, all authorised users can easily delegate authority to somebody else (see also Sandstorm, 2015). Miller et al. (2003) discussed this and other issues in detail, and traced many of the computer security problems that still exist today back to the use of ambient authority in ACL systems, which impairs the safe delegation of authority. However, the dynamic, ad-hoc and safe sharing of resources is a prerequisite for collaboration.

The aforementioned door key analogy is easy to grasp, but has issues of its own. The most obvious one being the revocation of authority once granted. In this real-world analogy, the only way to remove authority would be to replace the whole door lock. This is often not realistic or desirable. Nevertheless, many contemporary token-based methods of accessing web services belong to this category. Although keys have their uses as capabilities, a different kind of capability, referred to as object capabilities, solve this issue by offering richer semantics.

2.2. Object capabilities

The concept of object capability proposed by Dennis and van Horn (1966) parallels the concept of object-oriented programming (OOP) in many aspects. An object capability is a reference to an object, and represents both the authority and the means to access a resource. The authority is exercised by sending messages to the referenced object. The messages that the object understands make up the authority it represents.

In software systems, an object usually represents some real or abstract resource encapsulated by an interface (the messages that the object understands). For our purposes, objects exist within (computer) programs on machines connected by networks. Thus, a capability, aka a reference, to an object might span program and machine boundaries.

In any case, possessing a reference to an object equals the authority to access it. According to the object capabilities concept, the owner of an object reference is able to share it with somebody else by sending them a message (Fig. 3; Granovetter, 1973; Miller et al., 2001), which in our case depicts the relationships of computational objects (capabilities) in a network and their change over time. Fig. 3 recalls the original communication between Alice, Bob and Carol. In our case, all three actors are objects. Alice has a reference (a capability) to Bob, and is therefore authorised to send the message foo to Bob. This message includes a reference (a capability) to Carol, authorising Bob to talk to Carol as well.

2.2.1. A revocable object capability

Object capability systems operate under the assumption that the only way to communicate is through object references. In other words, if Bob receives a capability to Carol, then Bob has all the authority that this capability entails. One of the common objections against capability systems is that once-issued capabilities cannot be revoked. In ACL systems, the operation and authority of revocation is external to the user who is supposed to access the resource; for example, an administrator can remove a user from the ACL of a particular file or change the access rights to the file. Object capability systems allow new objects to be created that instantaneously forward all messages they receive, facilitating the creation of a valid work-around (Fig. 4). In this case, Forwarder (F) is such a capability that Alice sends to Bob. Optionally, F could even have a different, perhaps limited (read instead of read/write) interface to the one Carol offers, weakening the original capability to Carol. This alone would not yet allow Alice to control Bob’s access to Carol. Inserting another forwarder (Revoker R) under Alice’s control that forwards F’s messages to Carol solves problem. R’s interface can include a revoke message, which cuts the connection between R and Carol, effectively revoking Bob’s access to Carol. Bob can still talk to F, and F will still forward all messages to R, but the communication line is broken at R. As a result, Alice has revoked Bob’s authority to talk to Carol.

2.3. Implementations of capability security and object capabilities

Considerable research has been undertaken on capabilities, and different applications have been developed, ranging from operating systems (EROS, 1999; KeyKOS, 1988), programming languages (eights.org, 1998b) and web capabilities (Waterken, 2009) to filesystems (Tahoe-LAFS, 2021) and a web application platform (Sandstorm, 2021). Burtsev et al. (2017) presented research on a capability-enabled cloud infrastructure to facilitate secure collaboration at the infrastructure level. While it is desirable to secure the underlying infrastructures using capabilities and a least authority approach, the simulation infrastructure we propose operates at a different level, just below the application layer. From the perspective of safe collaboration, the Sandstorm platform is particularly interesting. Its goal was to create a web-based self-hostable productivity suite and platform with an app marketplace that allows untrusted applications to be run together in a cluster environment. The whole system was built on capability principles to allow secure communication among untrusted apps and users. The basis for Sandstorm was the Cap’n Proto protocol.

![Fig. 3. Three owners of an object capability (Alice, Bob and Carol) communicating via the message foo (Miller et al., 2003).](image1)

![Fig. 4. Construction of a revocable forwarder, through which Alice revokes Bob’s authority to communicate with Carol (Miller et al., 2003).](image2)
A decentralised contemporary simulation infrastructure needs to support a variety of systems and languages. Cap’n Proto is a data interchange format and capability-based RPC protocol, which allows disparate systems to communicate via object capabilities with an eye to efficiency and performance. Implementations of Cap’n Proto protocols are available for many different languages, enabling the integration of a wide range of existing systems. Even though Cap’n Proto is not as widely used as other pure RPC protocols (e.g., gRPC, 2021), it has proven to be stable and is also used in production systems, for instance at CloudFlare (Cloudflare, 2021). Capability security, speed, efficiency, platform and language agnosticism suggest that Cap’n Proto is a good foundation for a distributed simulation infrastructure.

2.4. A simple example of a Cap’n Proto API

As a data interchange format, the Cap’n Proto protocol specifies the structure of messages being transferred from a sender to a receiver. These messages are strongly typed, but not self-descriptive. For this reason, Cap’n Proto defines a special schema language that is used to describe the message structure. A compiler is then used to create code and data structures in the target language, allowing messages to be manipulated.

Cap’n Proto schemas contain two broad categories of structures, data structures and interfaces. The data structures represent the layout and type of data being sent. Interfaces are the entry point to the capability-based RPC system, and represent runtime objects and the messages these objects may receive. Messages in Cap’n Proto implementations are always sent asynchronously and – like a method call in an object-oriented

Fig. 5. An example of the use of a capability to a climate service to acquire a capability to a time series (object) at a particular Lat/Long coordinate (A), then restrict that time series to a 30-year range (B) and finally obtain the actual data for the time series (C).
language – can return values (data structures or other interfaces). Due to its asynchronous nature, the return of a value is depicted in the example below (Fig. 5) as a separate message, directing back from the receiver to the sender of the original message. Conceptually, this is the same as sending a message to the receiver, with an additional parameter representing the sender. Once the result is ready, the receiver can return it to the sender.

In the example realm of environmental science, Fig. 4 shows the use of a simple remote climate service to access data of a time series at a particular geo-location of the period 2021–2050. We assume that our user possesses a reference (a capability) to a remote climate service that holds these data. When the service is ready, it will deliver a time series capability back to our user (Fig. 5A). This capability is authorised to send all messages to a time-series object that the user understands. The simplified interfaces for the Climate Service and the Time Series are shown below (Fig. 6).

Words in bold represent capabilities (interfaces), words in italics denote data structures or simple types, and underlined words depict the message names that an object implementing this interface understands. The Cap’n Proto schema language allows the use of many of the structures often found in (object-oriented) programming languages such as inheritance, nested structures and unions.

After receiving the time series capability, the user only holds a reference, and the potential to acquire the data or query metadata. Since our user only wants a 30-year data subset, he sends the subrange message to the time series capability to receive another time series capability representing that 30 year subrange. This process is illustrated in Fig. 5B. In contrast, Fig. 5C shows the user sending the final data message to the just-received sub-time range. The result now is pure data, the structure of which has been described in the schema above as List(List(Float32)). The documentation of the schema would describe it as a list (the days of a list) of floating point values (the day’s data). The order of the day’s data, for instance, is defined by the header message available in the TimeSeries interface.

A particular point to note is the seemingly inefficient interface that TimeSeries offers. In order to obtain the required time series data, the user had to do three full network round trips – three requests to the remote interface. To optimise this, other interfaces (Fig. 7) could be possible, e.g.

To do the job, a ClimateService object could understand only one data message with three parameters revealing the location and time range requested, which then requires only one network round trip. An obvious problem in the long run could be the proliferation of the messages and message combinations an interface needs to declare that all needs are met, e.g. a version of the retrieved data within a certain time range, but with a different set of climate elements. The original API design did not have this problem because common object-oriented principles could be applied as network round trips, albeit seemingly at the cost of latency. A more serious problem is that in this case, the POLA has been violated: the second API version has no way to prevent a user from possessing this capability to acquire all data offered by the service, an authority we might not want to give the user. Traditional RPC systems have to work around this problem by adding a further layer of user management or actual capability-like access tokens (as an additional parameter to the message) to the system. In our API, the user is free to restrict authority at any level in which a capability is involved. If really necessary, our user could also just have received the capability in Fig. 5C to the sub-time range from somebody with higher authority, e.g. a co-worker. Cap’n Proto (and Cap’TP) solve the network inefficiency by using a concept called promise pipelining (erights.org, 1998d). The user of an API can immediately continue sending messages to the data structures and to the capabilities received as a result of previous messages. Sending a message in Cap’n Proto will return an object (a promise) that represents the future result. This promise will resolve to the actual result whenever the remote side is ready and the result has been received. In our example, the following Python code (Fig. 8) could express Fig. 5A–C using promise pipelining.

Only when the Python `wait()` method is called on the returned promise to the actual data will the Cap’n Proto implementation send the requested message. This means that only one network request is made, and all three messages are sent simultaneously. The subrange message can be delivered whenever, on the remote side, the first time series’ promises are resolved. When the sub-range time series is finally available, the data message can be delivered and the actual data returned to the sender. Note that, depending on the computing process, also referred to as a vat (erights.org, 1998c), in which these capabilities are active, these messages (subrange and data) can almost be plain local method calls within the remote system. Nevertheless, the mechanism is completely flexible and network-transparent because either of the returned capabilities from `closestTimeSeriesAt` and/or `subRange` could be remote to the vat in which the climate service object itself exists.

Concluding, the above Cap’n Proto API can retain the advantages of a proper domain API, support capability security principles, and still be similarly efficient as an efficiency-optimised custom RPC API.

3. Towards a model and simulation infrastructure

In the previous sections, we gave the reasons for creating a decentralised model and simulation infrastructure, and described which means of communication are necessary within the infrastructure to allow secure collaboration between data, models and users. In the following, we attempt to outline the structure of such a simulation infrastructure and showcase its envisioned use with a few simple examples.

While Cap’n Proto APIs can also be used to implement the mechanics of running models, access data or couple models, we initially focus on the creation of a coordination API and a set of simple, mostly general, services. The presentation in this paper leaves the implementation of these services consciously open. They can differ greatly between use cases, and may comprise different methodologies, ranging from single threaded scripting (for example in Python) over data and work-flow-

**Fig. 6.** Simplified Cap’n Proto schemas for remote Climate Service and Time Series interfaces.

**Fig. 7.** A Cap’n Proto interface description for an optimised Climate Service, merging multiple options into the data message.

**Fig. 8.** Python code snippet illustrating promise pipelining by interacting with future results and only one network interaction on the last `wait()` method call.
based approaches to highly parallel implementations in languages such as Go (2021).

In the first step, we incrementally create a coordination API and simple general services. Essentially, we can identify four different domains that an API has to cover.

**Administrative interfaces** to the infrastructure, which allow the management (e.g. addition/removal) of resources (models, data).

**Model and data interfaces** to allow access to different services representing a wide range of data and models. Such a service represents an instance (or instance factory) of a model, set of models or dataset. Given a capability to the service, the user or a program can control the service according to its available interfaces. For models, the obvious needs are to run it, either completely or stepwise (e.g. day by day), with a given configuration. Some of the functionality will be similar to specifications such as OpenMI (Harpham et al., 2019), which may be a way to bridge into external environments and runtimes. We expect these APIs to be deployed primarily by users requiring maximum flexibility, either via scripting or dedicated graphical user interfaces to data and models. Capabilities to data will then practically evolve into parameters to model runs, meaning that they never need to be accessed or transferred to the user’s computer.

**Simulation interfaces** offer a higher-level concept for the assembly of models and data ready to be applied to a certain problem domain. Rather than having to care about individual model instances, users of these APIs will focus on the use of previously created sets of ready-to-run workflows, which we call simulations. Later, further simulation APIs might allow the interactive creation of simulations.

**Scenario interfaces** that complement the simulation interfaces acknowledge that, at the top-most level, simulations are always used to explore scenarios. These interfaces allow the creation, manipulation and easy access to scenarios, and simulations bundled/configured with particular sets of data.

While the simulation APIs (backed by the model and data APIs) manage the technical aspect of how a simulation is to be performed under defined boundary conditions (hardware, software, available data, etc.), the scenario APIs reflect the management and application of these simulations.

The overarching goal of these APIs is to create a composable set of services. Since we abstract all resources via object capabilities, we can write programs that interact with these remote objects. The actual programs can become additional services extending the API, as long as they implement a Cap’n Proto interface. This enables the organic and incremental creation of new services by anyone anywhere. Comparing this to a single system and a single programming language, it is similar to object-oriented libraries that are dependent on other libraries for their implementation. Systems like the Java platform or.NET are considered powerful because they offer a huge set of functionality ready to be used and extended by composing library objects into new, potentially more complex libraries.

### 3.1. Administrative interfaces

The administrative interfaces within the infrastructure can be grouped into two categories:

I. **Interfaces for managing the infrastructure**: this is usually the task of dedicated administrators using special APIs implemented by graphical user interfaces to manage the actual infrastructure.

II. **Interfaces for managing the administrative part of services**: Here the view becomes more blurred. These APIs actually belong to the service domain and are used to administer a service. The domain and the task govern who is authorised to do this. For example, a service will usually have at least two faces, one accessed by the general public and one to configure and maintain the service itself.

The following simplified example shows how a service can be accessed by two or more capabilities, depending on the task or users involved.

When a service is first launched, possibly via the administrative API, it returns one or more persistent capabilities (sturdy refs) to the service. If a user possesses one of these sturdy refs, they can connect to the service. Fig. 9 below shows the Cap’n Proto schema of a general registry service, which, when launched, returns a reference to the actual service (Registry), a reference to the administrative interface (Admin) and a reference to a Registrar interface. As a result, three levels of authority have been introduced, mediated by three capabilities.

The registry service allows other services to be registered. A user can browse the registry and acquire a capability to a service needed for a particular task. This capability can then be used to execute the task: run a model, retrieve data, etc.

Our example above supports freely configurable categories, which are used when registering a service. Categories could be called “climate”, “soil”, “model” or “crop management”. A user who possesses a Registrar capability can use the register message to register their own service at the registry.

The definition of categories permitted for a registry instance is mediated by the Admin interface. A user who possesses a capability to a remote object with this interface can add and remove categories.

The Registry interface is what users are expected to work with, either at the API level using a programming language or in an abstract way using a graphical user interface. Users can send the message supported Categories to get a list of the valid categories or the message entries to obtain a list of all entries on the basis of the category name supplied.

Once a user/program has obtained a capability to a required service, it can be used directly or queried for a sturdy ref. The latter allows direct connection to the service at a later time, meaning that the registry is no longer needed.

From an administrative user’s point of view, starting a service registry at the command line might look like this (Fig. 10):

The started service creates a number of persistent capabilities (sturdy refs) which allow connection to the service. The URI for the above sturdy refs can be read as follows:

```cappn://… the protocol specifier insecure or public key of the service … to connect securely to an encrypted service @host:port … the IP or hostname of the server and the port number being used /a-sturdy-ref-token … a secure unguessable token to identify the actual remote object being referenced
```

```
interface Admin {
    addCategory @0 (category : Text) -> (success : Bool);
    removeCategory @1 (category : Text) -> (success : Bool);
}
interface Registry {
    supportedCategories @0 () -> (categories : List(Text));
    struct Entry {
        category @0 : Text;
        ref @1 : Capability;
    }
    entries @2 (category : Text) -> (entries : List(Entry));
}
interface Registrar {
    register @0 (ref : Capability, category : Text);
}
```

**Fig. 9.** Simplified Cap’n Proto schema for a service registry – split into three separate interfaces for proper rights management.
3.2. Model and data interface

The model and data interface represents the lowest layer of the user-facing part of a model and simulation infrastructure. These services are supposed to be used directly by end users, and by the simulation interface layer above. Moreover, we expect these services to be the building blocks of other more specialised services or model aggregates and model pipelines.

Fig. 11 illustrates the use of the model as well as the data interfaces. The diagram assumes that the user has already obtained capabilities to a climate time series, a soil profile and a set of management events (via existing data services) as well as a capability to the instance of a crop growth model via a model instance factory. The user then sends the run message to the model instance capability, supplying all three data capabilities as parameters; the user finally receives the model run’s result as pure data.

As can be seen in this and the previous examples, everything boils down to the ingredients: capabilities, messages and data structures. Contrary to traditional RPC protocols, capabilities are first-class, meaning that they cannot just be the receiver of a message, but also the contents of a message and the result of a message sent. This property allows the composability of services and enables the simulation API to build on the model and data interfaces.

3.3. Simulation interface

The simulation API addresses the problem that, for many use cases, the actual model and data services are too low level. The services are the building blocks for a simulation at the next higher aggregate level. A simulation may be the application of an agroecosystem model to a larger region to simulate crop yields, involving the allocation of many model instances and data sets. While Fig. 11 illustrates running a model on a single climate time series, a simulation for a larger region looks similar, but involves a capability to a climate dataset. In this case, a climate dataset is a collection of time series used to acquire capabilities to time series at particular locations within the simulated region. How a simulation is then actually configured, often depends on the implementations, e.g. scripts using MPI (The MPI Forum, 1993) or MapReduce-style (Dean and Ghemawat, 2004) pipelines.

Finally, Fig. 12 describes a more complex example at the level of the simulation API. The figure also illustrates the flexibility offered by Cap’n Proto for a cross-infrastructure setup, as mentioned in the introduction. In our example scenario, User A shares the authority to read data from Infrastructure A (first step) with User B. User B runs Simulation X, which needs an instance of Model B and read access to Datasets A and B (second step). After running Simulation X, User B wants to fix a perceived problem with Version 1 of Model B, which has only been used and deployed in Infrastructure B. She finds and fixes the problem on her local development machine and tests the bug fix with an instance of Model B running there. Instead of running Simulation X with a capability to Model B in Infrastructure B, she runs the simulation with Version 2 on her local laptop (third step).

3.4. Scenario interface

Scenarios can be considered as having the same relationship to simulations as simulations have to models/data. While a simulation is an aggregation of models and data at the technical level, scenarios move closer to the actual application domain. The setup of a simulation should address all the technicalities of the models and data to be used. The setup
must be flexible enough to cater for different models and datasets. Scenarios usually involve running a simulation with different configurations, e.g. including different climate datasets or crop management strategies. The organisation of the different runs belongs to the realm of the scenario API. The scenario API is technically the same as all other layers, except that the interfaces and their implementations change, reflecting the domain to be described.

4. Discussion

4.1. Barriers to scientific collaboration

Scientific collaboration in environmental modelling has grown in recent years. Climate systems modelling started long ago to build simulation models that consist of various modules, each representing physical components and related processes of the global climate system (Manabe and Wetherald, 1967; Phillips, 1956). Advancements were made along two lines: (i) the integration of additional details and components (Bennartz et al., 2013; Bony et al., 2006; Sun and Hansen, 2003) and (ii) an increase in the spatial and temporal resolution (Iles et al., 2020; Jones et al., 1997). The development of climate models, and later earth system models (Friedlingstein et al., 2006), always centred on a computing and data storage infrastructure and, in turn, contributed to fuelling the further development of supercomputers and data storage technology (Palmer, 2014). Observation infrastructures (Giard and Bazile, 2000; Reichle et al., 2010) that continuously feed large data streams into these facilities were subsequently added. Today, a number of large environmental research institutions develop and host their own modelling systems. Applications of their models for short-term weather forecast or long-term climate scenarios, for instance, are collected in an ensemble approach to assure the most robust prediction as possible. The Coupled Model Intercomparison Project (Eyring et al., 2016; Meehl et al., 1997) is a prominent example from the field of climatology. This idea was also adopted in the impact modelling community (Schellnhuber et al., 2014), including hydrologists (Maxwell et al., 2014), economists (Robinson et al., 2014) and agronomists. In the latter case, the Agricultural Model Intercomparison and Improvement Project (Rosenzweig et al., 2014) brought together modellers to compare their models in how they represent the interaction of crop plants with their soil and atmosphere environment, simulating changes of crop yields, water consumption, nutrient losses, greenhouse gas emissions and other variables of interest. In this case too, the ensemble mean of a population of models proved to be the best predictor of many target variables (Martre et al., 2015; Palosuo et al., 2011). However, it turned out to be very difficult to improve such models, because there were no common standards for code development in this community; single modelling groups invested heavily in clipping out code from models to insert it into other models in order to test the consequences of the newly added set of algorithms on the prediction of a target variable (Maiorano et al., 2017; Wang et al., 2017). At the same time, much time and effort has been invested in developing common standards for interfaces (Buahin and Horsburgh, 2018; Evans, 2012; Jöckel et al., 2005) or data formats (Porter et al., 2014), in a bid to run models developed elsewhere under the roof of a common infrastructure. What all these efforts have in

Fig. 12. An example of a cross-infrastructure setup. Step 1: User A possesses Data A in Infrastructure A, and shares this data with User B. Step 2: User B runs Simulation X using the data received from User A and Model B (available at Infrastructure B). Step 3: User B reruns Simulation X, but with Version 2 of Model B on User B’s laptop.
common is that components, models or data need to be transferred to a new host organisation, which then becomes responsible for (temporarily) maintaining and running them (Knapen et al., 2020). The success of such efforts, measured by the number of items involved and used for creating end products (reports, studies, publications), seems to be moderate, and strongly dependent on the commitment of the hosting person or group. If complete models are transferred, the problem often arises that the new host institution does not have the capacity to engage with model recalibration or code adjustments, if required by a new research question. This again hampers a straightforward simulation pipeline with the external models involved. A distributed infrastructure would allow for such constructions. For instance, the models could remain temporarily with their developers, and run on a common infrastructure elsewhere. On a more granular scale, model components could also be handled in the same way, allowing for joint model development with distributed process components (Midingoyi et al., 2021).

4.2. Organic growth

A range of large collaborative model developments are under way in various science communities related to environmental research, the largest probably being the Community Earth System Model (Hurrell et al., 2013), which involves a number of sub-component development teams. The concept builds on component-based software engineering, and software libraries that are developed peripherally and can be shared across multiple institutions. Host institutions have been specifically designed and equipped to meet the host function in the best possible manner, including massive financial support from governmental level. Although the idea began as a grassroots effort, it now relies on a major institutional commitment to keep the development along the desired trajectory. This example has worked so far, and produced a large number of research products that have a high impact on the community, especially in the context of global climate projections and the understanding of the underlying concert of coupled processes. However, many more communities are under way that are less visible, and consequently less heavily supported, because their work does not address the most prominent societal challenges. In such cases, grassroots efforts may remain as such for a long time, and require lower thresholds for collaboration, especially if they are embedded in a co-creation process with stakeholders (collaborative modelling; Ulibarri, 2018). For such research efforts, easy access to powerful computing facilities as described above will surely be an advantage, allowing them to grow organically on the ground of great ideas, but with limited resources.

4.3. The building and maintenance of large remote object graphs

Unlike an object-oriented program, whose analogy we used further above, the actual application running is (a) not under the central control of a single computer/process/developer and (b) all components of such distributed programs are subject to failure risk. Thus, while it is already difficult to create bug-free programs in traditional environments on a single device, a distributed application has to deal with all the possible issues that may arise for multi-threaded concurrent applications, including network-related issues and the lack of central control over all the components. While the object capability paradigm and its implementation offers the means (consequent asynchronous promise-based concurrency) to create such an infrastructure, it is still a challenging undertaking. One particular problem is that of how to keep an existing remote object graph (the set of remote services connected via capabilities) stable in the advent of a failure. If a service goes offline (network outage, server reboot, software crash), it has to be restarted automatically. At places where many services are hosted and people rely on their availability, some kind of supervisor system will be required to assure this automatic response. Mechanisms to design such fault-tolerant systems have already existed for decades and are applied widely in production, as exemplified by Erlang-based systems and OTP Supervision Trees (Sloughter et al., 2019) or the current world of container orchestration, e.g. with Kubernetes (Valim, 2019).

A further challenge is the evolution of software and services that are reliant on existing services. The existence of persistent references to objects/services within other systems is meant to allow the creation of new services that are reliant on a potentially large network of other services or remote objects in that network. In the case of traditional software, updates are released at some point, restarting its runtime object world. In a decentralised system, the evolution of parts of the system requires migrating to newer versions of a remote service, moving services to other physical locations or removing services that are considered outdated, while keeping, if possible, clients unaffected (a similar challenge arose for clients connected via the HTTP protocol). Strategies have to be developed for all these cases. Starting small, organic growth is expected to allow the incremental and agile development of these strategies along with the evolution of the distributed system.

4.4. Relationship to contemporary data and computational infrastructures

The object capability approach presented here is considered a flexible layer on top of more foundational infrastructures, allowing to ignore particular implementations of data storage representation and remote services execution. As such, the resulting distributed object graph can represent any imaginable API beyond our example from the environmental realm. In any case, the underlying infrastructures must be of computational nature. Unless Cap’n Proto APIs for acquiring these computational resources themselves are created, the provisioning of actual hardware is completely external of the concepts presented here. In most cases, capabilities will allow to treat the runtime as an implementation detail, no matter if a remote object is living on a personal laptop, a HPC node or a virtual machine provided in a cloud. The same applies to data infrastructures which facilitate efficient storing, accessing, synchronisation and sharing of potentially large amounts of data. Often, infrastructure providers expose APIs on their own to allow automated access and retrieval of data. In context of the capability approach, it means that capability-based APIs can be created, which directly use the exposed APIs of these data infrastructures. However, the process of data management is often an external independent institutional process and outside of the considerations in this paper.

4.5. Range of applicability and high-performance considerations

Many high-performance applications need to access large amounts of data, thus the question arises if and how distributed computing can fit into this picture. In general, the object capability concept is well suited to control and coordinate tasks. This allows to abstract access to data and resources and thus increases the likelihood for interoperability between systems. It also allows to build further abstractions on top of lower ones to ease the infrastructures use. On the other hand, any added layer of abstraction adds a layer of indirectness and thus most likely reduces efficiency.

The most glaring problems are related to data locality and latency. To process large amounts of data, code has to have as much bandwidth to the data as possible, often facilitated by having the data physically close to the code processing it. Consequently, it makes a huge difference whether data is living on a remote system across the internet, on a high-performance network back-end, on a slow hard-drive, or in an in-memory or in-processor cache. As a user of a capability does not have to know where the remote code is actually executed, this approach is potentially less suited for bandwidth-critical applications. However, up to a certain point the capability approach allows increasing the available bandwidth by moving a remote object closer to the data in need to be processed. While in such case the abstraction introduced by the Cap’n Proto layer may lose some of its efficiency, it renders some use cases possible that did not exist before.
Latency in the communication between two distant objects scales with distance and with the number of indirect layers being involved. It becomes problematic if two models are to be tightly coupled and thus demand a large amount of communication between them. Similar to the bandwidth problem above, it makes a big difference if the code of the two models to be coupled is located within a single thread, process (multiple threads), machine (multiple processes), node, network, or within two machines across the internet. In all these cases, remote objects are interoperable, but at some point latency will limit the practicability and the concept becomes useless. As demonstrated in Fig. 12, the potential lies again in the emerging opportunities: the user of Model B was able to complete the task in Step 3 because to the infrastructure the instance of model B was just another capability, which allowed her to run version 2 on her laptop. If the simulation would have involved some tight coupling of model B to some other code, it may have resulted in a fairly slow simulation, but at least it would have worked. In the end, it all boils down to a balance between performance and flexibility, an underlying trade-off discussion that also dominates code development. It remains to be seen how much ground can be covered in practice when applying the remote procedure call methodology for high-performance computing, for which it certainly poses some challenges.

5. Conclusion

In the light of heterogeneously distributed computing facilities, models data and skilled personnel, we have highlighted the barriers that currently hamper scientific and engineering progress in the field of environmental modelling. We have argued that collaborative scientific work would achieve faster, more efficient progress, and described in very general terms how a capability-based decentralised model and simulation infrastructure could facilitate such collaboration efforts across individual institutions. While existing or future designs and implementations of similar systems remain valid, we have offered an alternative view on the interoperability of existing and newly built infrastructures. Starting with a simple set of service abstractions for models and data, the creation of simple user interfaces will enable support for these services and the creation of programs and scripts that directly access data and models.

In recent years, we have seen powerful examples of making data remotely available, and it has become evident that we need continuously improving tools to work on these remote interfaces. Web-based interfaces developed for this purpose already facilitate a large range of operations at this stage. However, they lack the property of composability and a decentralised method of managing security. Although any single deficiency can be worked around, as a whole these concerns hinder seamless interoperability and thus collaboration, at both the software and human level. We have highlighted the existence of alternative approaches that act inclusively and have the potential to unite disparate efforts in a self-organising way.

Software availability section

- Languages: C++, C#, Python, Go, Java

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References


MEA, 2005. Ecosystems and Human Well-Being. Washington DC, USA.


