Execution State Management in Named Function Networking

Christopher Scherb
Dept of Mathematics
and Computer Science
University of Basel, Switzerland
Email: christopher.scherb@unibas.ch

Balázs Faludi
Dept of Mathematics
and Computer Science
University of Basel, Switzerland
Email: balazs.faludi@unibas.ch

Christian Tschudin
Dept of Mathematics
and Computer Science
University of Basel, Switzerland
Email: christian.tschudin@unibas.ch

***Abstract*—This paper introduces a mechanism to steer long-running in-network computations over ICN, or more specifically, over an NFN network. NFN (Named-Function Networking) is an extension of ICN that performs in-network resolution of expressions instead of mere content retrieval of a single name, as in ICN. One problem NFN faces are timeout events which are not discriminative enough to distinguish between healthy and ongoing computations from blocked threads or from connectivity problems. Moreover, we would like to change parameters of a long-running computations on the fly, to fetch intermediate (approximate) results or to simply cancel/stop/resume a computation. To this end we introduce Request-to-Compute messages (R2C) which generalize keep-alive messages. In this paper we report on experiences with a running prototype (steering a n-body simulation) and relate R2C to PubSub and Rendezvous protocols.**

I. Introduction

In today's Internet, cloud computing has become a dominant topic: Data storage and computation power are offered as services and are seamlessly bundled with networking functionality. Especially for small devices with little storage and computation power it is beneficial to use external resources, be it the remote cloud or nearby edge servers.

Information Centric Networking (ICN) [1] is an alternative network pattern to current IP networks. It shifts the focus from host-centric IP solutions towards fetching content objects via names in a location-independend way: Provided a simple name, the network can decide how and from which location a content object is delivered. Named Function Networking (NFN) [2] is an extension to existing ICN networks that enables an ICN to deliver not only data that was already published but also on-demand results of computations, whereby the network can optimize *where* a computation is executed. NFN orchestrates computations for a specific user at the edge of the network as well as for services inside the network [3].

Our first NFN implementation focused on how to en- and decode a computation inside an ICN name, how to forward interest message with a computation inside the ICN name, how to optimize the execution location and later on how to secure results [4]. One of the discovered shortcomings of using plain ICN Interest and Data primitives was that there was no solid mechanism to prevent computations from timing out. Moreover, once a computation was started there was no way to interact with it until it returned a result (which made it hard to debug NFN applications, for example). We address these problems with the introduction of "request-to-compute" messages (R2C). This paper describes the mechanics of R2C messages and shows broad usage for such messages for moving NFN even closer to the cloud.

ISBN 978-3-901882-94-4

II. Background

A. Information Centric Networking

Information Centric Networking (ICN) networks are based on two types of messages: Interests and Content Objects. While content objects store the actual data, interests are used to request a content object. To secure the available content and to prevent users from manipulated content, every content object is signed by the producer. To fetch a content object, users express a interest messages and forward them to an ICN node they are connected to for further processing. The ICN node itself consists of three important data structures: the Content Store (CS), the Pending Interest Table (PIT) and the Forwarding Information Base (FIB). When an ICN node receives an interest message, it first checks the CS, whether the available content is already cached. If it finds a matching content object inside the cache, the interest will be directly satisfied by replying with the content object. If there is no matching content object in the CS, the ICN node will check the PIT. The PIT is used to deduplicate interest messages and to store a trace to the user who expressed the interest message. If there is a PIT entry available, the interest message will be appended and not be forwarded, since the message has already been forwarded once. If there is no PIT entry available, the ICN node will lookup the FIB to forward the message. Thereby, all interfaces with a matching entry could be used concurrently, but only the first reply message will be processed. After transmitting an interest message, a node performs several retransmits, to ensure, that the message was not lost. If no answer is received in a specific timeout interval, the PIT entry will be deleted, and no content object will be returned. When an ICN node receives a content object, it checks if there is a matching PIT entry. If a PIT entry is found, the node will reply to all nodes which requested the content object, otherwise the content object will be deleted.

B. Named Function Networking

ICN usually retrieves content that was beforehand made available for retrieval. NFN extends the capabilities of ICN by allowing to request content through "recipes" based on named functions and named parameters. Thereby, content retrieval becomes a special case of functional programming in NFN. NFN interests request a computation (or its cached result) in form of a λ-expression. Such an expression can reference other content objects and portable functions to be applied to them. The network is responsible for obtaining the required content, optimizing the execution location based on resource availability and mobility, running the computation, and returning the results to the client. In NFN the concept of thunks was introduced to prevent timeouts. Thereby, a client could ask for an estimation on how long the computation may take and the timeout intervals of the PIT entries will be adjusted accordingly.However, if the computation requires more time than the estimation, the result cannot be delivered anymore.

III. Controlling Computations in NFN

In the past, our NFN implementation was only able to start a computation and to wait until it finished. Since the computation was running inside the network, a user had no access to some sort of "Process Control Block (PCB)" as it exists for example in UNIX. Furthermore, since a user only received the final result, remote-debugging a computation was very tedious.

Request-to-Computation (R2C) messages are an extension to NFN which enable the user to access the PCB and to interact with it. Also, the R2C messages can be used to *request intermediate results*: The main use case is to obtain the current state of long running computations (e.g. simulations) but also for fetching debug information. In the following we present our approach to controlling and managing network computations in NFN using R2C messages.

A. Request-to-Computation Messages (R2C)

When accessing or changing a computation in the network, there are mainly two things to specify: The identity of the computation to be controlled (whose potentially multiple execution sites are unknown to the requestor) and what we request from it. We map R2C messages to ordinary ICN Interest and Data messages as we have already done with NFN messages. In Figure 1 we see an outer, long-lived NFN-request that starts the computation and returns the final result. Inside this transaction there can be multiple short-term R2C exchanges that enable to steer the ongoing computation.

Technically, ICN interests are flagged as being R2C messages by an additional name-component

```
/somePrefix/call <funcname> <params>/
            R2C/<command_and_params>/NFN
```

By relying on longest prefix matching—which is common in ICN networks—it is possible to add the NFN (and R2C) suffix without affecting the routing. The prepended routable prefix and the computation expression identify which computation is addressed while additional name components at the end can be used to specify the command and parameters that "the computation" should receive.

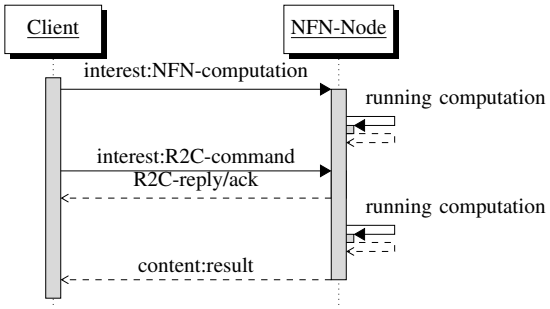

Fig. 1. Schema of NFN with R2C-messages

R2C messages are forwarded the same way as ICN or NFN interests. Thus, we use the same mechanisms (i.e retransmits) to prevent packet loss. If there nevertheless is a packet loss, the same user driven recovery is required as for any other ICN packet. Every R2C message is confirmed by a ACK message.

B. Request-to-Computation Commands

In the following we give an overview of implemented R2C commands and how they operate.

1) *Start/Stop* *:* The simplest command is to start a new computation. This function was already available by sending am ordinary NFN-interest

```
call <funcname> <params>/NFN
```

Additionally, to be consistent with the R2C commands, a computation can now be started using the interest:

```
call <funcname> <params>/R2C/start/NFN
```

Both interests have exactly the same behavior. After having started a computation—especially if it is a long running computation—a user may realize that the result is no longer needed and could be stopped. This is achieved through the stop command:

```
call <funcname> <params>/R2C/stop/NFN
```

When such an interest message is received, the computation is aborted and there is no way to restart the computation again. However, the results of subcomputations which have already been completed will still be available inside the network's cache. If a computation is stopped while waiting for the results of subcomputations, the subcomputations also have to be stopped (using R2C messages, too). Furthermore, it is possible that multiple users request to start the same computation. In this case NFN pools the requests and the computation is only executed once. This is achieved by adding all incoming interests to the PIT. To prevent that a computation is stopped by one user while other users are still waiting for the computation to terminate, the computation is not stopped until all PIT entries are removed. Thus, a stop message does not abort the computation directly, but only removes the corresponding PIT entries if appropriate.

2) *Pause/Resume* *:* Pausing a computation can be useful for stream processing, which can also be considered as a long-running computation. In case the user pauses the video, it does not make sense to continue the computation since the partial results of the conversion only stay inside the network's cache for a limited time:

```
call <funcname> <params>/R2C/pause/NFN
```

To continue the video the user resumes the computation:

```
call <funcname> <params>/R2C/resume/NFN
```

If there are multiple PIT entries for a result of a computation it is required to fork the computation when a pause request is received. Thus, it is possible to satisfy the pause request which returns a unique handle while not affecting other requests. The PIT entry for which the pause request was issued will be changed to point to the forked computation state and will be paused until a resume command is received. After receiving the resume command, the corresponding interest will be satisfied by the forked computation.

3) *Get Intermediate Result* *:* During long-running computation, intermediate results are a way to fetch the current computation state. Therefore, an additional function is added to NFN: A NFN developer can explicitly publish an intermediate result via the executing NFN node. These intermediate results are identified by ascending numbers. To fetch an intermediate result, a user has first to learn whether there is already data to be fetched, which can be done with the method

```
call <funcname> <params>/R2C/
       intermediateResultAvailable/NFN
```

This method returns a number X that identifies the latest available intermediate results and permits to fetch it. To fetch an intermediate result the parameter X is given to the R2C function `getIntermediateResult`:

```
call <funcname> <params>/R2C/
       getIntermediateResult X/NFN.
```

The function `intermediateResultAvailable` delivers live results to the user. When called multiple times, it could return different results. Therefore, it does not fit to the common ICN principles of immutable data, since it acts as latest version service. To avoid conflicting versions of the same data, the data created by `intermediateResultAvailable` are not cached. This exception is reasonable, since the function `intermediateResultAvailable` only delivers meta data, about which intermediate results are available, while on the other hand the function `getIntermediateResult` provides concrete data objects, which are immutable.

By writing the current state of the computation as intermediate results, a user has the possibility to fetch information about the computation, which are required to debug an in-network computation. Thus, writing an intermediate result which contains the current state of the computation is similar to setting a breakpoint in a debugger.

4) *Timeout prevention* *:* Timeouts in the PIT lead to deletion of the reverse path: If a long-running computation exceeds that PIT timer, no results will be returned although it was computed.

We address this problem by introducing R2C keep-alive messages. Shortly before a timeout would occur, a R2C keep-alive message is issued:

```
call <funcname> <params>/R2C/
    keep-alive/NFN
```

If the compute server receives a keep-alive message, it will check whether the computation is still being executed and if it is, the keep-alive message will be answered with an empty content object. The PIT entry will only be deleted if no response is received to a keep-alive message within the common timeout interval. Usually, the node closest to the client will issue the first keep-alive message. All other nodes on the route to the computation node will see the keep-alive messages and update their PIT, without sending own keep-alive messages.

C. Controlling Fan-out

Forwarding a request for a NFN result can lead to multiple computations being started if there are multiple FIB entries which match the same prefix. Starting a computation on multiple nodes is a waste of computational resources. Additionally, the client has no information about how many nodes are running the computation and it cannot verify that all computations were notified by an R2C message. To address this in the case of duplicate FIB entries, the NFN forwarding engine can be optimized to choose one of them and only if the computation fails and no result is received (this can be monitored by using the keep-alive-messages), the other entry is selected. To find the running computation, it is required to track the forwarding state and attach the selected FIB entry to the PIT. As soon as the computation is notified, the downstream entity will receive an ACK message.

IV. Validation through Prototypes

In order to understand the impact and usefulness of the proposed R2C messages we implemented prototypes for use cases in three different areas: a physics n-body simulation, a PubSub application and a Rendezvous protocol supporting server and client mobility.

Our goal was not to perform any benchmarks (comparing the performance of our system with similar applications) but rather to verify functional correctness and versatility. All test runs were using the simple environment shown in Figure 2.

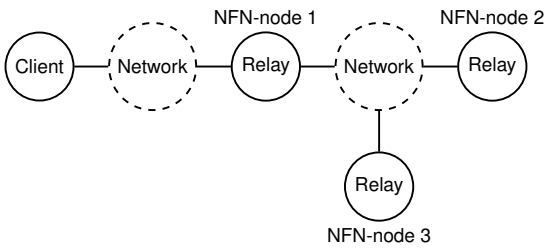

Fig. 2. The simple environment used during our evaluation.

A. Physics Simulations

Simulations are usually high performance applications– in our case a physics n-body simulation where particles subject to gravity float in space and eventually collide, forming bigger particles and ultimately stars. We chose this kind of simulation because requesting the state of the simulation at any point of the computation gives a meaningful result (that can be visualized). Moreover, intermediate results can be used to checkpoint the intermediate state of the computation. Finally, it is a perfect case for exercising the R2C's keep-alive messages.

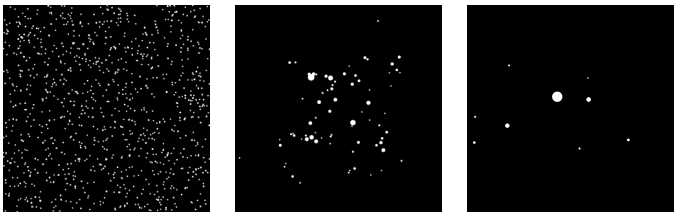

Fig. 3. Different intermediate states of our n-body simulation

An n-body simulation computes the position of particles based on gravity, movement and the previous state. Given a matrix of the position, weight and velocity of a certain number of particles at t_0 the n-body simulation computes the positions, weights and velocities at t_1. If two particles collide, they will be merged into a new particle with the joint weight and mean velocity of the collided particles. After computing t_1 out of t_0 it is possible to use t_1 to compute t_2. Every state t_n is stored as an intermediate result. Thus, the user can access all past states of the computation.

After the first step is computed, it is possible to fetch the result, to verify that the computation runs correctly and to visualize the result. In fact we could visually verify that the computed results are meaningful as is shown in Figure 3. To this end we defined a NFN computation that creates an image out of a state t_k. Combining the simulation function with the visualization part demonstrated at the same time function chaining with intermediate results: Assuming (in Fig. 2) that NFN-node 1 stores the `visualize` function, NFN-node 2 stores the input data and NFN-node 3 stores the `simulate` function, we can express the interest message

```
i1=call /visualize (call
    /simulate /firstParticleState m)
```

where m gives the number of iterations that should be computed. Figure 4 depicts the workflow. First, the interest is forwarded to `NFN-Node 1` where the outer computation can be executed. Next, the inner call

```
i2=call /simulate /firstParticleState m
```

is forwarded to a location where the input data file `/firstParticleState` is stored (`NFN-Node 2`) and the function `/simulate` is fetched with interest `i3` from NFN-node 3 (not visualized in Figure 4), so that it is possible to start the inner computation. Meanwhile, the outer computation awaits intermediate results. As soon as they are available, it fetches the intermediate results and creates an image. Therefore, it is required to periodically check if an intermediate

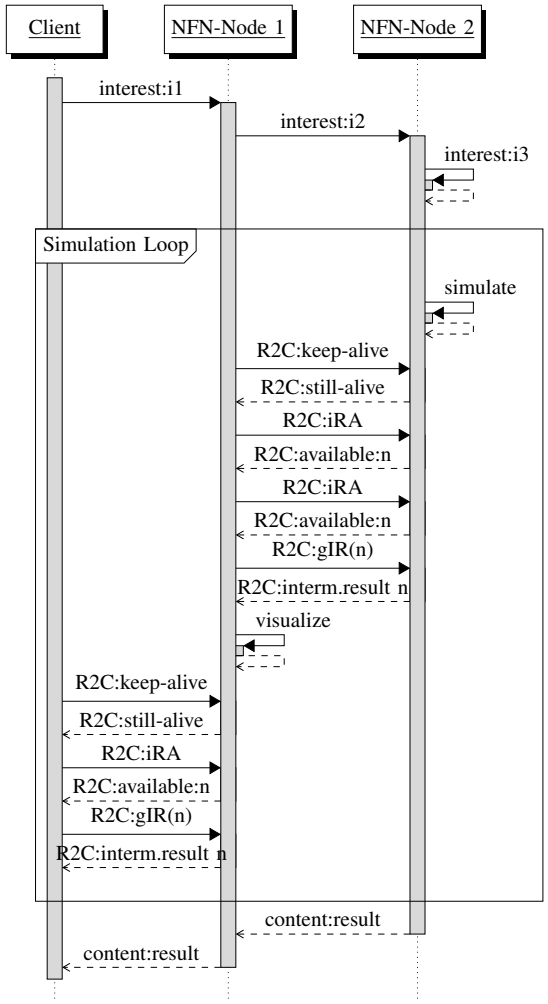

Fig. 4. Long-running n-body simulation in NFN using the R2C extension. NFN-Node 1 stores the function `visualize` and NFN-Node 2 the input data file `firstParticleState`. Interest `i3` is used to fetch the function `simulate` from NFN-node 3. `iRA` stands for `intermediateResultAvailable` and `gIR` for `getIntermediateResult`. n is the number of the latest available intermediate result. The simulation loop runs for a specific number of simulation steps, m in our example.

result is available. The image is returned as intermediate result to the user (who also has to periodically check if a result is available). When the m-th state is computed, the Simulation Loop will be left and the last result state is returned as the final result. The outer function `visualize` is stopped as soon as the final result of the inner computation is received and the last image is computed and sent to the user. During the computation, it is required to use timeout prevention (keep-alive messages), since the computation requires more time than the typical timeout interval.

B. Publish/Subscribe

Publish/Subscribe (PubSub) is a common pattern to distribute irregularly published content to multiple receivers. In the following we demonstrate how long-running computations and intermediate results can be used to express the PubSub

pattern. PubSub can be seen as middleware which informs a user if a new result or dataset is available. Thereby it is not required that the consumer and the publisher are directly connected or available at the same time. In NFN, a publisher can start a computation which will act as a kind of middleware and "buffer". The computation periodically checks if there is new content published and fetches it as soon as it is available. New content is stored as an intermediate result. As a subscriber goes online, it checks if there are new intermediate results available. Since NFN uses the PIT to pool duplicated interests, multiple consumers can subscribe to the same content while the "middleware computation" is only running once. Note that each consumer pulls items at its own pace by requesting intermediate results with next number needed for its sequence. To unsubscribe, a consumer simply stops the computation.

Mapped to the test scenario in Figure 2, the client (subscriber) issues a PubSub computation which will be executed on NFN-node 1 while NFN-node 2 operates as the publisher.

C. Rendezvous protocol for Terminal Mobility

ICN networks naturally enable (consumer) client side mobility: After changing location, a client can just retransmit an interest message to continue fetching data. However, there is no native mechanism for (producer) server mobility in ICN. One way of supporting mobility is by using a non-mobile waypoint e.g. rendezvous extension for HIP [5]. The server registers with its current prefix/address on the waypoint node. Every time the server changes its location, it will update the entry on the waypoint node. Thus, the client can ask the waypoint to get the server address and to request data. The disadvantage of this mechanism is that special infrastructure/middleware is required. As already shown in the previous example, NFN with R2C messages can be used to replace common middleware. In our test scenario, in order to offer a service, a mobile server (NFN-node 2) will start a computation which runs on NFN-node 3 and creates intermediate results with the current prefix (address) of the server. Since the latest intermediate result should contain the current prefix/address of the server, the server has to update the computation on NFN-node 3 every time it changes it's location. To do so, the computation awaits a message from a server. This message can also be a R2C message (see Section V-A). Using the intermediate results we avoid a single content object containing the server address. Thus, the data objects itself remain immutable, while only the function `intermediateResultAvailable` contains mutable data (see Section III-B3).

By requesting the latest intermediate result, the client can fetch the current server address to request the actual data. It is also possible that the client creates its own waypoint-computation to enable bidirectional mobile data transfer. This mechanism could be used to create peer-to-peer applications.

V. Extended Usage of R2C Messages

In the following we will present further extensions to the R2C system to show the capability of this concept. Additionally, we show how existing concepts can be implemented by using NFN without changing the network nodes itself, based on the extensions we propose.

First we show how R2C commands could be used to influence a computation during runtime.

Next we discuss a extension to the intermediate results to make them more flexible and to fit requirements application of some application. Later, we propose a way how to implement the concepts of segments and versions with the R2C extension.

A. Dynamic Programming

In some cases, a user starts a computation and later on decides to change some of its parameters, based on intermediate results. This can be achieved by structuring the computation into runs where the controller fetches the result of a run, decides how to continue and then starts the next run. Using R2C messages (and without having to resort to self-contained runs, but instead referring to existing execution state in the network), the developer of a long-running named function would insert blocking stages at the end of a run: The computation would be put on hold, waiting for the intermediate result to be fetched and waiting for a subsequent resume command. By sending the R2C command

```
call <funcname> <params>/R2C/continue(
    <new computation> )/NFN
```

the execution is resumed by using the instructions (and parameters) from `<new computation>`.

B. Fetchable Names for Intermediate Results

Previously, we focused on numbers to identify intermediate results. Thus, when requesting an intermediate result, a number was given to the function `getIntermediateResult` as parameter. A further extension to the intermediate results is to publish the intermediate results by using an ICN name. Instead of the number, a NFN function developer could also specify the name of the intermediate result explicitly to better match requirements of the applications. In this case, a `intermediateResultAvailable` request has to return a list of all available results instead of the number of the latest intermediate result and the parameter of `getIntermediateResult` will be the name of the intermediate result that should be fetched. Additionally, it the name could be used to fetch the result directly by using this name instead of calling `getIntermediateResult`.

C. R2C commands for "strategies"

One further use case of R2C commands (which we did not implement, yet) would be to transfer a running computation to another provider or to influence how to distribute a computation in the network. This opens up the prospect of emulating "ICN strategies" by using NFN computations. For example, one could easily change e.g. the ICN forwarding or caching strategy by just deploying a different computation [6] i.e., the network's behavior is considered as long-running computation. Thus, the creation of overlay networks is possible as we shown in our PubSub example. It is also feasible to use computations to create networks for the usage in data centers, where special forwarding and replication strategies are required [7].

D. Segments as Partial Results

In ICN it is common practice to split large content objects into segments. A segment is identified by a segment-id, either stored in a name component or in meta data while the last segment is marked by using a last-segment identifier [8]. This way it is possible to transmit live streams even if the last-segment id is unknown when the transmission is started. In the context of NFN, a segment can be seen as a partial result and would be handled by the mechanics of retrieving intermediate results: The consumer can reconstruct the original large content object by combining all partial results. That is, the producer splits the content in multiple intermediate results instead of segments. Since intermediate results are numbered, comparable to segments, the translation of names is straight forward. Similarly, the R2C command `intermediateResultAvailable` replaces the last-segment identifier. Note that this is not a 1:1 mapping (since in classic ICN the last-segment information is a marker in the name of the last segment while in our R2C case we have to use `intermediateResultAvailable`), but the overall behavior remains the same.

Some ICN implementations use catalog files (also called manifests) to describe the segmentation of data. if required, a pointer to the next catalog file (this property enables streaming by using catalog files). A user first requests a catalog file which contains pointers to the actual segments and, Since intermediate results with R2C are enumerated and require special calls to `getIntermediateResult` also indicating the name of the original computation, it is not possible to directly use such numbers in catalog files. However, using explicit ICN names for intermediate results, as explained in Section V-B, one can again populate catalogs without having to refer to the computation that created these segments.

E. Versions as Provisional Results

Versions and R2C semantics also work well together: Versions are a natural way of looking at intermediate result. For example, the accuracy of simulation results increases with every iteration, leading every time to an "improved result".

We can use R2C commands as a replacement of the classic ICN version field meant to work around the cryptographic binding between a name and the chunk's content: New content leads to a new name with a changed version field. In classic ICN, the new version is indicated by an increasing integer number or by a timestamp that follows the segment identifier. The monotonically increasing version number can be directly mapped to the numbering of intermediate results (and the R2C command `intermediateResultAvailable` enables users to discover the latest version of a data object). To deal with timestamps, it is required to know which timestamps are available. By explicitly naming intermediate results (see Section V-B) `intermediateResultAvailable` will return a list of intermediate results. If the names of the intermediate results contain timestamps, this becomes a list of all available timestamped versions. In other words, versioning with timestamps can be mapped to the R2C concept of intermediate results, too.

F. Iterators

While ICN segment and version fields are packet-level concerns, the higher-level picture is that the network should provide support for iterators at the level of programming languages. This doesn't come as a surprise given NFN's functional spirit: The R2C commands are generic enough to support iterating both along the segment dimension (including streaming of data) as well as over the versions of a specific chunk. We have not yet explored this mapping to a high-level programming language, though.

VI. Conclusions

In this paper we introduced the concept of Request-to-Computation (R2C) messages to manage the state of a NFN computation. In previous NFN versions it was only possible to start a computation and wait for a result. The R2C extension enables users to control the computation when it is executed inside the network. This is not only required to abort computation, but also for debugging, timeout prevention, fetching intermediate results and client-side computation steering. The R2C extension can also be used to express classic ICN concepts like content segmentation or chunk versioning in a novel way, potentially even replacing them.

In order to validate the R2C concept, we applied it to a physics n-body simulation (to verify intermediate results and timeout prevention), a Publish/Subscribe scenario (to use computations as in-network buffers) and a mobility waypoint (to keep a mobile server's connectivity state).

References

[1] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '09. New York, NY, USA: ACM, 2009, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/1658939.1658941

[2] M. Sifalakis, B. Kohler, C. Scherb, and C. Tschudin, "An information centric network for computing the distribution of computations," in *Proc. of the 1st International Conference on Information-centric Networking*, ser. INC '14. ACM, 2014, pp. 137–146.

[3] H. Zhang, Z. Wang, C. Scherb, C. Marxer, J. Burke, and L. Zhang, "Sharing mhealth data via named data networking," in *Proceedings of the 2016 conference on 3rd ACM Conference on Information-Centric Networking*. ACM, 2016, pp. 142–147.

[4] C. Marxer, C. Scherb, and C. Tschudin, "Access-controlled in-network processing of named data," in *Proceedings of the 2016 conference on 3rd ACM Conference on Information-Centric Networking*. ACM, 2016, pp. 77–82.

[5] J. Laganier and L. Eggert, "Host identity protocol (hip) rendezvous extension," 2008.

[6] C. Scherb, M. Sifalakis, and C. Tschudin, "A packet rewriting core for information centric networking," in *Consumer Communications & Networking Conference (CCNC), 2016 13th IEEE Annual*. IEEE, 2016, pp. 67–72.

[7] D. Mansour and C. Tschudin, "Towards a monitoring protocol over information-centric networks," in *Proceedings of the 2016 conference on 3rd ACM Conference on Information-Centric Networking*. ACM, 2016, pp. 60–64.

[8] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, k. claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, "Named data networking," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 66–73, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2656877.2656887