NDN Technical Report NDN-0074. http://named-data.net/techreports.html
Revision 1: May, 2021

NSC – Named Service Calls, or a Remote Procedure Call for NDN

Daniel Meirovitch, Lixia Zhang
UCLA Computer Science Department

Abstract

Named Data Networking (NDN) is a potential new internet architecture which has gained momentum due to its ability to enable applications in scenarios where the current IP networks have struggled. Among these scenarios, mobile ad-hoc networks or delay tolerant networks. Existing applications and systems make heavy use of remote procedure calls, or RPCs, to enable more complex applications. Designing a framework for remote code execution in NDN could ease development of applications and systems in NDN. Named Service Call (NSC) answers how to securely process client requests, and timely inform the client of their results within an NDN architecture.

1. Introduction

Today's internet has a variety of content types. A large share of today's most popular web applications relies on dynamic content generated by executing code on a remote machine. Remote Procedure Calls (RPC) are not new technology and have been used since the beginning stages of the internet. RPC began by using XML to encode communications and provide semantics so that code can executed regardless of the operating system. XML is very verbose compared to JSON, so eventually, JSON became the common way to encode RPC communication. Many web applications began to use RESTful frameworks which borrow many of the RPC concepts for executing remote code with a given set of parameters.

REST [1] was the dominant paradigm for a long time, but now gRPC [2] and GraphQL [3] are additional new ways to think of remote code execution. gRPC improves performance by using protocol buffers and binary encoding, along with making it easier for developers by machine-generating client libraries. GraphQL adds more flexibility to resource requests based on relationship structures. These techniques have become essential for developing applications and systems.

NSC is a proposal to bring a Remote Code Execution framework to NDN. NSC considers potential use cases for complex NDN applications and answers how they can securely and effectively communicate between machines.

The main use case considered will involve smart cameras placed around a city which share videos with a central cluster for analysis. This use case is common in major cities for surveillance and traffic checking. We investigate this use case because it stresses some of the main challenges of remote code execution. Specifically:

- How to protect the remote executors from processing garbage data which wastes CPU and Memory resources.
- How to protect the sensitive input and results (video frame data and any ML Detection).
- Sending video frame data as input is a network intensive task.
- It may take the central cluster a variable amount of time to process certain requests.
- The central cluster would require many machines listening and processing requests in parallel due to the data requirements.

These security, parallelism, and functional challenges are common to many applications but do not have a standard way of being handled in NDN currently. NSC combines learning from previous research to propose a remote code execution framework that considers these challenges.

The primary contributions are in the NSC design. NSC's first contribution is to create a communication pattern for sharing input and retrieving remote results, which relies only on communication tools already implemented in NFD. NSC uses three Interest-Data transactions to match an overall request-response pattern. The overall pattern will be familiar to most other developers and is highly compatible with existing NDN networks.

The proposal also secures the communication pattern to protect both sides of the transaction. Signed Interests at the start protect the system from any abusers, and encrypted data protect the sensitive input and results. Both security techniques are already common in NDN, to ease implementing NSC into any NDN applications.

NSC is also inspired by previous works and proposes a mechanism for handling results that take a long time to compute. Some applications require long-lived code execution, and these requests may not fit well into typical network timeouts. In NSC, the Caller is informed of any delays and where/when to retry for data. This application-level solution is preferable to network-level adjustments.

Finally, we must consider use cases that require complex and distributed application architectures. So, we discuss how an application can work with NSC to provide fault tolerance for remote code execution.

We also implemented a smaller sample use case, which sends credit card numbers for verification, as a proof of concept for the NSC framework.

Throughout this paper, we will examine these contributions. In Section 2, we will look at the motivation and specific challenges that any NDN Remote Code Execution framework must answer. In Section 3, we will review previous works in this area that have informed NSC. In Section 4, we will discuss the NSC design details. Then in Section 5 and 6, we will review the current and future states, and finally conclude in Section 7.

2. Motivation

Remote Code Execution is a critical feature for new applications and systems, but NDN does not currently have a standardized model for how to execute remote code. Different applications or systems developed with NDN define their way to communicate. The lack of standardized solutions makes it difficult for software engineers new to NDN, since they may not have the expertise necessary to create an application and communication patterns. Proposing a flexible model for remote code execution in NDN, using Named Service Calls (NSC), may open future use cases.

In traditional point-to-point IP networks, RPC works using the request-response communication pattern. shown in Figure 1. The client establishes a bidirectional communication channel to a server. Then the client will send all the input to the server. The server acknowledges receipt, and then the client will wait until the server eventually sends the results.

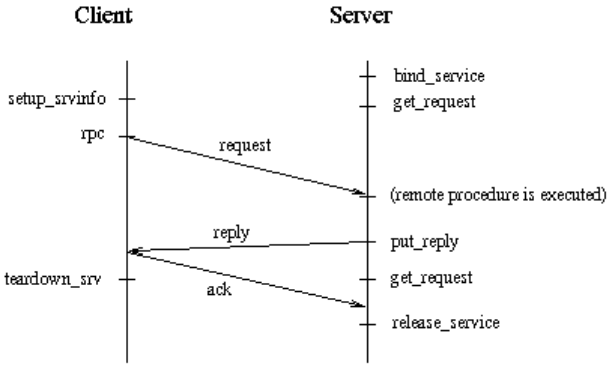

Figure 1: Current IP-Based RPC Model [4]

There are some variations in RPC behavior in IP networks. Some tasks which take a long execution time, such as VM creation on Cloud Services, may only return a task handle at the start. The user must periodically check the status of their original request using the task handle. This flow has useful applications which informs our NSC design in Section 4.

To implement RPC in NDN, a naïve solution would make the RPC request an Interest packet, and the response a Data packet. However, data is generated dynamically based on the request parameters and that creates new challenges for NDN. These challenges were presented earlier in the Introduction and can be generalized as:

1. Servers may handle computationally intensive tasks, so they need to protect themselves from rogue clients.
2. Clients and Servers need to protect sensitive data, such as inputs and results, from unauthorized access.
3. Clients need to rely on Servers without worrying about the internal architecture details that they may use to provide high availability.
4. Clients need to send potentially large amounts of input data, but NDN interests need to be small.
5. Clients need to know how and when to retrieve their results, without a reliable bidirectional channel like TCP.

Our NSC Proposal attempts to answer each of these challenges.

3. Related Work

There have been previous works that tackled the need for remote code execution frameworks in NDN. They attempted to answer some of the same questions outlined above, and the NSC design is heavily inspired by their solutions.

3.1. RICE – Remote Method Invocation in ICN

RICE [5] proposed a general framework for ICN architectures that answers the challenges in Section 2.

- Servers authenticate Clients with a 4-way handshake. During this handshake, they exchange security information (such as passwords or keys) to facilitate encryption. After, Clients will possess a security token that they can use to invoke remote functions without repeating the handshake.
- Instead of sending input parameters directly in an Interest, Clients will publish their input parameters. Then as part of the 4-way handshake, Servers can learn how to retrieve the Client's input.
- At the end of a function invocation, Servers return a "thunk" to the Client. The thunk includes a name and a time estimate for the Client to retrieve the result. If the time estimate is incorrect, a revised estimate is sent back to the Client.

3.2. DNMP – Secure Network Measurement Framework for NDN

DNMP is part of a larger effort to create a set of libraries that could be used for distributed measurement of NDN network devices [6]. A publisher-subscriber (pub/sub) protocol is used to send commands to network nodes and receive

measurements back. The prototype DNMP answered the challenges by:

- Signing all commands and data while using NDN Trust Schemas to establish the security model.
- Clients publish RPC Requests as data, and any Server will learn of this new RPC Request via an NDN Sync protocol. In this case, they use the *syncps* protocol.
- After Clients publish RPC Parameters as named data, they subscribe to a result name. All Servers publish their results to the result name and the Client retrieves the results when ready. Clients learn of the results via an NDN Sync Protocol.

3.3. Lessons from Previous Works

Each previous work offers a different solution to the challenge of retrieving dynamic content, but there are valuable lessons to learn.

First, the Client should publish input parameters as named data, instead of directly including those parameters in the Interest packet. It is necessary to keep Interest packets below the 8 kb max size, and large input parameters may affect Interest packet size. As another benefit, publishing input parameters makes them auditable and reviewable later.

The second, is that NDN security primitives are the preferred means for securing remote code execution. Trust Schemas and Signed Interests offer great flexibility along with their data security guarantees. There is a potential performance loss compared to token-based approaches like RICE [5]. However, DNMP [6] showed the ease of use for NDN primitives, which is important for adoption by other application and system engineers.

The third is how the use cases can influence communication patterns. In DNMP, NDN Sync Protocols are used to update Clients and Servers about new data or commands for remote execution. However, the DNMP use case is specific to network management where nodes publish a command for all other subscribed nodes to execute. NSC needs to handle the use case where Clients only need a single result from a single Server. NSC also needs to be prepared for other scenarios such as a significantly distanced Client and Server (edge camera to central cluster) or a high number of Clients (edge cameras placed throughout the city). These scenarios would not be able to use a sync group, like in DNMP. Because current sync protocols are limited by how fast and far they can propagate updates.

So, in the absence of a sync protocol, the Client needs some way to learn when the Server results are finished. Whether due to difficult tasks or Server compute failures. Network layer solutions, such as extending the Interest lifetime, can cause severe performance degradation. Interests do not guarantee reliable delivery, so a lost Interest with a 60 second lifetime would not be detected for 60 seconds. Instead, NSC needs a dynamic system to keep the Client informed of the results processing.

4. Design

This section will review each aspect of NSC's design. Specifically, we will how examine the design attempts to solve the main challenges of remote code execution in NDN.

4.1. NSC Design Assumptions

NSC makes several design assumptions which are common to other NDN applications. The first is that there may be a distributed and redundant application architecture. Clients may be far away from the servers, and there may be many servers listening for requests. So, clients should be able to work with any server.

Considering the distributed application architecture, NSC also assumes that all servers are reachable. The servers make routing announcements, which creates routes for clients to reach them via Interest anycasts. If servers control their own reachability, that means they can potentially share a name.

Finally, NSC assumes that all entities in the system have gone through the necessary NDN bootstrapping process. That means all entities have obtained trust anchors, their own certificate, and security policies as trust schemas. These are all necessary components of any NDN application, and NSC will rely on them for its own security assumptions.

4.1. NSC Overall Communication Design

The NSC traffic flow is visualized in Figure 2. We rename the Client to NSC Caller and Server to NSC Executor, since at different moments they switch roles in the traditional NDN definition of Client and Server. Each number and description below correspond to a numbered step in Figure 2.

1. The Client (NSC Caller) sends a notification interest to the Server (NSC Executor).
    a. This notification interest is signed to prove identity.
    b. The notification interest contains an Interest parameter for the location name where the Input Parameters can be retrieved.
    c. The notification interest can optionally include a forwarding hint. In case the Input Parameter location prefix is not well known.
2. The Server (NSC Executor) responds and acknowledges the notification interest. This response includes:
    a. The name where the result will be stored.

b. An estimated time when the results will be available.

3. The Server (NSC Executor) sends an interest to retrieve the input parameters based on the name provided in step 1b and receives the data back.
4. The Client (NSC Caller) sends an interest for retrieving the result data.
5. If the data is not yet ready, the Server (NSC Executor) responds with the name where to retrieve the result from and with new timing estimates.

These steps meet our goals, and we will examine each one in more detail.

4.2. NSC Security Design

NSC's security design mainly aims to secure Servers and Clients from malicious attacks. The main security challenges that were identified earlier are restated below:

1. Executors may handle computationally intensive tasks, so they need to protect themselves from rogue Clients.
2. Callers and Executors need to protect sensitive data from unauthorized access.

One of the lessons learned from the DNMP is that NDN's security primitives, such as trust schemas and signed packets,

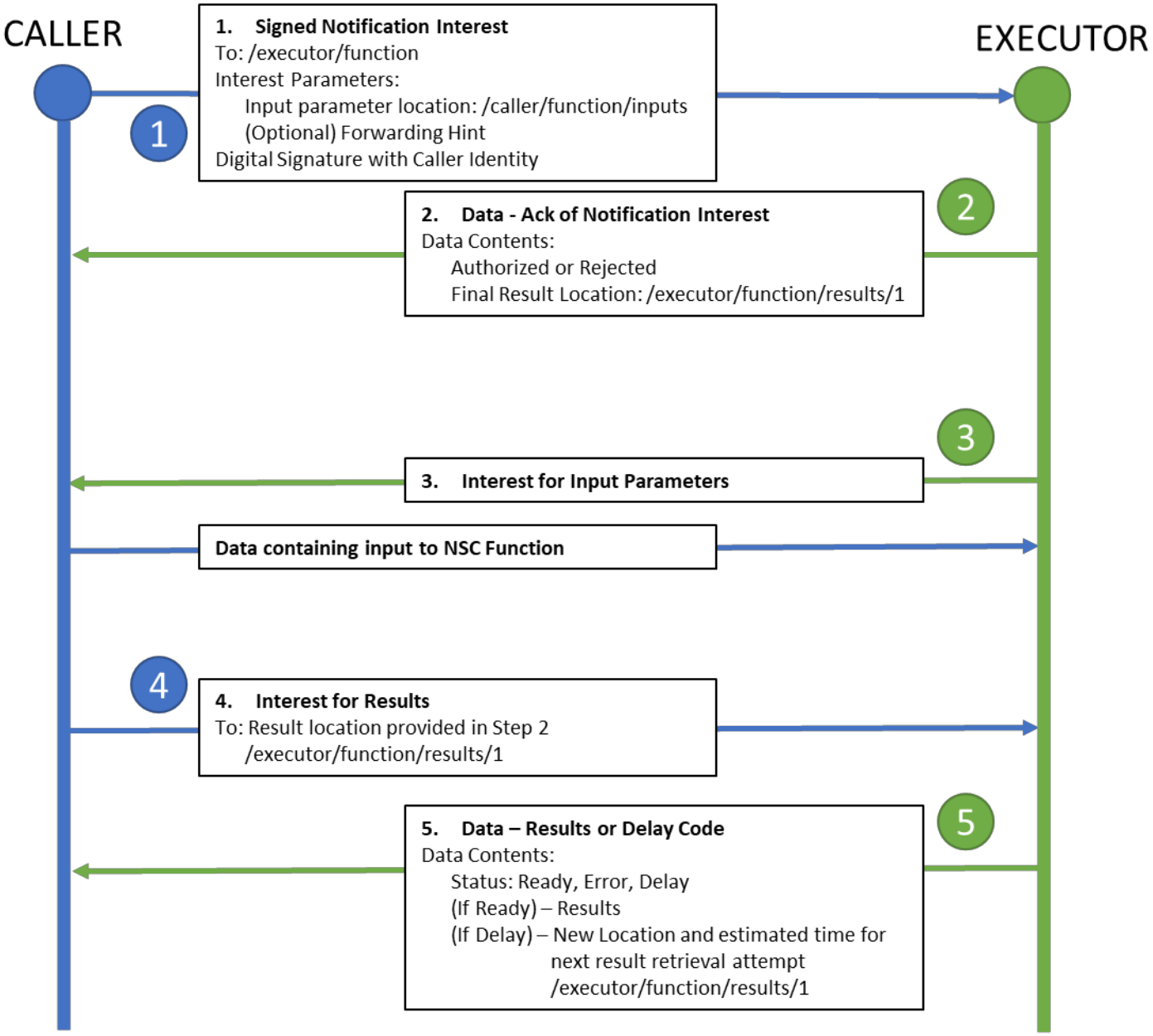

Figure 2: NSC Proposed Design and Communication Order

are easy to use for application developers [6] [7]. So, NSC addresses these challenges with these techniques.

Without security, an executor can be overwhelmed by computing garbage interests. For example, in our use case, unauthorized parties could send executors a high number of fake interests of video frames for processing. The central cluster would use all available resources and could no longer process valid data traffic.

So, to protect the Executor's compute resources, NSC requires the first interest to be signed. The Executor then verifies the source of the request before any function execution.

The next issue is about protecting the actual data. In our use case, video frame data could include private information such as personal movement data. Similarly, results may contain sensitive information. So, we need to protect the data when published at the NSC Caller and Executor for retrieval. Otherwise, any malicious actor scanning names or intercepting traffic along the forwarding plane could have unauthorized access to the data.

As a solution, NSC encrypts all data exchanges. The Caller encrypts the input data in step 1B before publishing. The Executor encrypts its result data in step 2b. So, the encrypted data is secure for the entire set of transactions.

Encryption can be handled in different ways depending on the use case. NSC supports more complex use cases via Named Access Control (NAC) [8]. These use cases can involve multiple callers and executors, in comparison to RICE which only supports a single caller and executor [5]. For example, NAC could make raw video frames only accessible to administrators, while the analytics results could be viewable by all traffic and law enforcement personnel.

The application owners who want to secure their data will need to provide a way for users and NSC Callers to enroll themselves into the service. This enrollment process includes out-of-band bootstrapping of trust anchors and the distribution of trust schemas. At this point, the NSC Executors would be able to authenticate the NSC Caller's initial Signed Interest. This is like enrollment and setup procedures that need to be done today for IP-based RPC clients. Such as Point of Sale devices at retail stores, or Cloud API Clients which require configuring identities and trust anchors. The exact trust schema used, like in most NDN applications, would then be variable on the application.

This security design addresses our challenges, although performance may be a concern. Verifying signed interests and their intermediate trust anchors can take time. However, this is not a unique problem for NSC, and all NDN applications should consider their trust anchors. In exchange, developers can easily incorporate NSC into their applications. Many of these security steps, such as encrypting data at rest, enrollment, and trust-schemas, are required anyways for a secure application. The new security challenge is protecting the Executor's compute resources. NSC uses signed interests, an already existing feature in NDN, to meet this challenge.

4.3. NSC Input and Results Location Design

The next two questions that the NSC design must answer are:

1. Callers need to send potentially large amounts of input data, but NDN interests need to be small.
2. Callers need to know how and when to retrieve their requested data, without a reliable bidirectional channel like TCP.

NSC again uses NDN fundamentals, namely Interest parameters, to carry this information.

The Caller first must publish their input parameters so they can be retrieved. Then they begin step 1 of the communication process and notify a given function hosted by the Server to begin executing. The Caller's signed notification interest will carry with it, the name of the published input parameters created just before. We choose to carry this information as an Interest parameter because its small data size will not have a large impact on the NDN forwarding plane. And the Caller is the one in the best position to alert the Executor to the location of the input parameters. An example of the content carried in this Interest Parameter can be seen in Figure 3. The exact formatting can vary depending on application choice.

Next, the Executor can use the information provided in the notification Interest to send its own interest and retrieve the parameters in step 2. At this point, the Executor will have already allocated a name for the final published results. The Executor will attach the final published result name as an Interest Parameter as well, as described in step 2.

Finally, the Caller can use the provided result name to initiate step 3 and retrieve the function results. In the next section, we will explore what happens if the results are not ready quickly.

The data names may not be in the NDN Forwarding Plane's FIB. So, in that case, a Forwarding-Hint can be included in the parameter field. The Forwarding-Hint is separated by a human-readable separator name which is surrounded by extra '/'. The extra '/' delimit the separator because two // in a row is an invalid name. So, our choice of separator names cannot be confused for any application parameters, and do not accidentally limit the application's namespace options. The Forwarding Hint in a more distributed use case could be an edge gateway for the Caller or Executor [9].

Ultimately both Caller and Executor are responsible for informing the other about the next name in the sequence. Only

after receiving the Interest carrying the relevant parameters, can they move on to the next step in the NSC sequence.

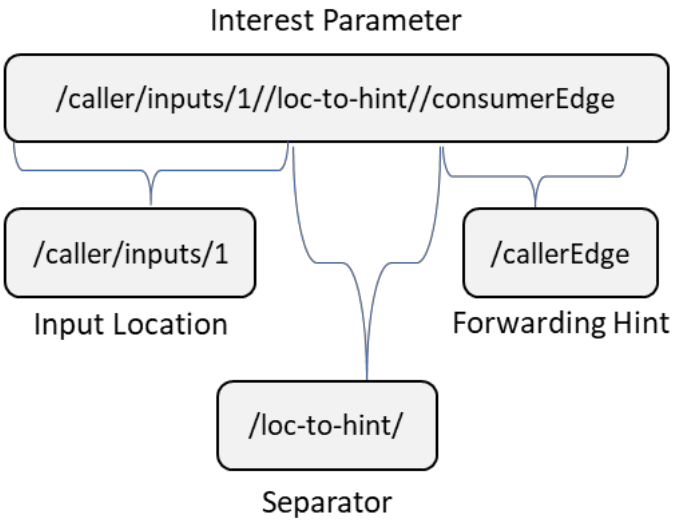

Figure 3: Interest Parameter Example

4.4. NSC Result Retrieval Timing

Most NDN applications today serve static content or content generated within a single Interest lifetime. However, some application requests may take much longer. For example, our use case requires sending a set of video frames for object detection analysis. NSC's design then must consider results that are ready quickly and results that take extra time.

The first case we consider is the simpler one, where the NSC results are ready within the lifetime of a single interest. In that case, we do not want to add a delay. So, the Executor will publish the named location in Step 2 and begin processing the input parameters as soon as it receives a data response. The Caller sends their Interest in step 3 for the result, and the Executor responds as soon as the result is ready.

A benefit of publishing the result location ahead of time in step 2, is that the Caller will not have any extra waiting before moving from step 2 to step 3. The Caller can send an interest immediately, and it will not be rejected. The Executor is already listening for that result name, while the Executor processes input data in a separate thread. So the Caller will receive results when they are ready and within the lifespan of the interest timeout, which defaults to 4 seconds.

The second case we consider is when the result data is not ready in time. The 4-second default timeout can be too short for processing large datasets. A simple solution would be to increase the Interest lifetime to a longer value. This may be set for specific transactions in an application that expects long processing times, or at the network level for all Interests. However, this naïve solution is risky. Primarily, if the Interest is lost, no node in the forwarding plane can recover until the timeout expires.

NSC uses a dynamic system to handle delayed responses, similar to RICE [4]. The Executor processes the result in a separate thread, and the Executor can check if the results are ready before the Caller Interest timeout expires. If not, the Executor will send a message indicating this state and the next location for data retrieval. The Executor and Caller can iterate as many times as required until the result is finally ready. NSC reuses the same name for the result data to maintain the idempotence of a single request.

The current implementation offers a naïve solution to notify the Caller to retry every 4 seconds if the result is not ready. However, a more complex application using NSC could estimate itself and provide more accurate wait times to the Caller. This timing estimate needs to come from the application code and may require significant statistics on past application behavior. The executor may even need to communicate with another system to retrieve a timing estimate. However, even with a simple timer that stays constant at 4 seconds (the default Interest timeout), we do not expect heavy congestion. That is because the Interest packet itself is small and does not contain any extra information at this stage in communication.

The application-level code is responsible for sending the delay message since the application is the cause of the delay. That puts the application in the best position to handle the issue. An example delay message is shown in Figure 4 that includes the key requirements: the delay status code (NOT-READY), the retrieval location, and the next retry time.

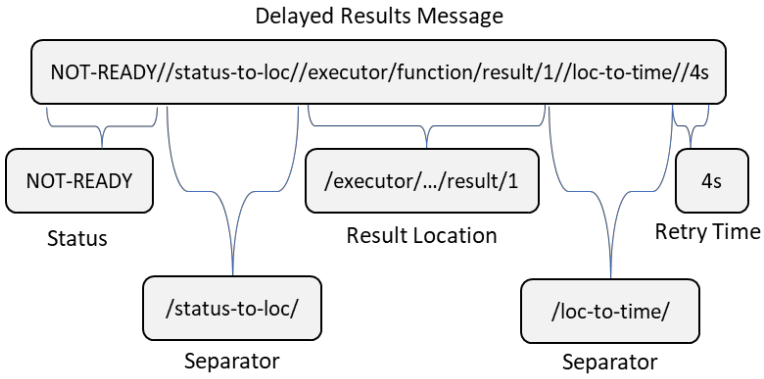

Figure 4: Delayed Results Message

4.5. NSC Scalability Design

NSC needs to assume that application and system engineers will want to create scaled-out and highly available systems for processing requests. Specifically, there may be many Executors listening in parallel to a shared name, some Executors may fail intermittently, and some tasks may have

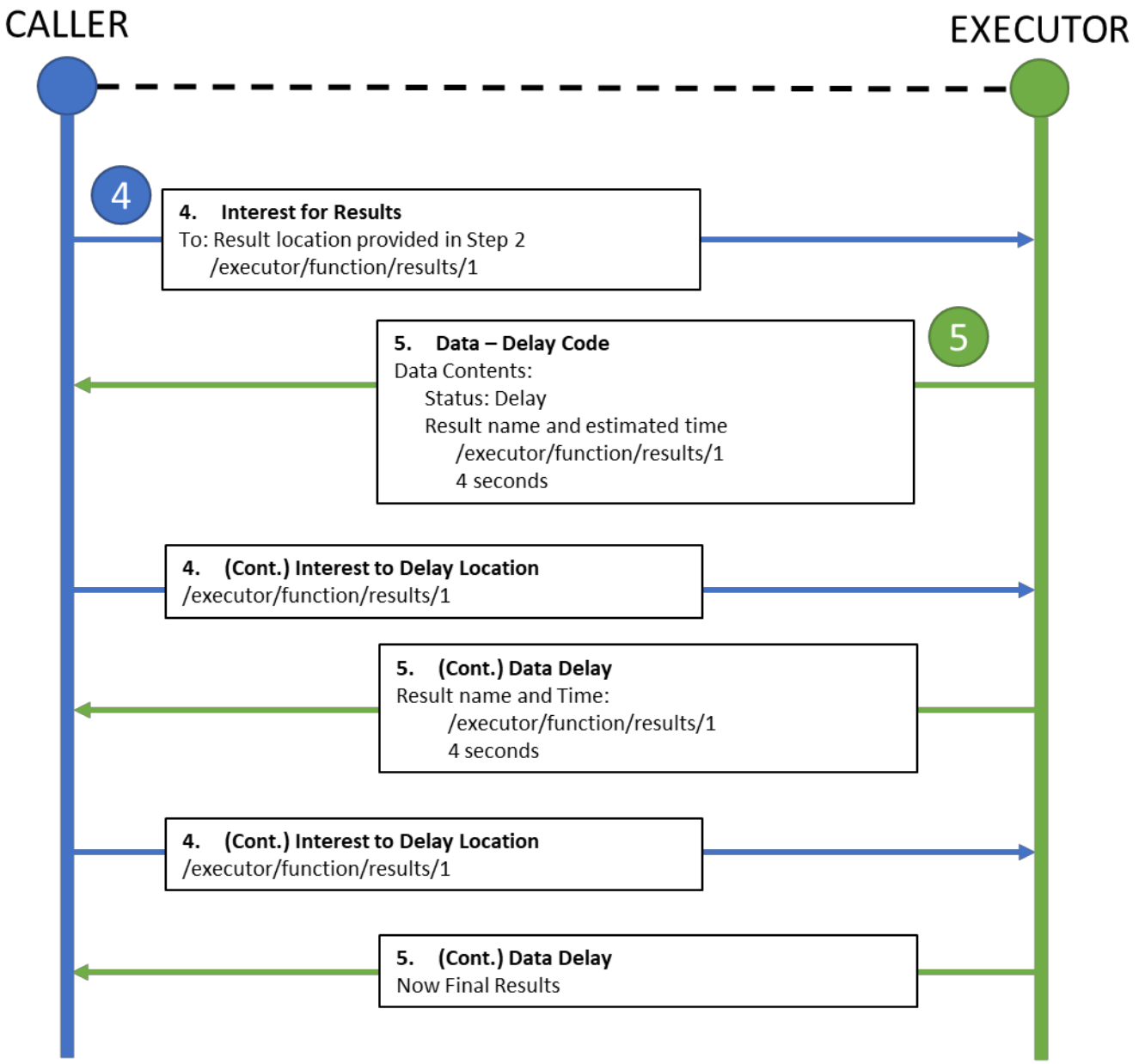

Figure 5: NSC Delay Handling for Results

nternal dependencies. Many internal architectures can satisfy these requirements, but the main impact to NSC is that we cannot assume a single Caller is speaking to a single Executor for the entire sequence.

To accommodate a scaled-out Executor infrastructure, NSC handles failures by assuming each transaction in the overall sequence is atomic. So once a transaction is deemed successful, that state is stored, and the Caller will only retry the current step as needed. We define a transaction as a single request-response at the application layer, which requires an Interest-Data exchange at the network layer. NSC has a total of three transactions as seen in Figure 6.

The below steps show how it would look in our sample use-case of video frame analysis:

1. The Caller initiates the NSC sequence by sending a Signed Notification Interest to a shared name, managed by multiple Executors.
2. One Executor processes the Interest and responds with a timing estimate plus result location if the Interest is authorized. Internally the Executor adds the

details of this task to a task queue to be picked up by itself or any other Executor later.

At this point, the first atomic transaction is complete. The Caller knows that the central cluster has received its request, and another node can pick up the sequence if that single Executor fails. All the necessary data has been received from the Signed Notification Interest.

The next transaction is to retrieve the input data:

3. An Executor picks up the task from the queue and sends a request to the Caller to retrieve the Input parameter.
4. The Caller responds with the input parameters, and the Executor saves the parameters to a distributed data store. It will then add a task to a pool to process the input parameters.

At this point, the second atomic transaction is complete If there is a network failure and the Executor never retrieves the input parameters within the Interest lifetime, then it can keep retrying. If a single Executor fails after sending the Interest, then the task is never marked as completed so another Executor can attempt the same task and retrieve the input.

The final transaction is to retrieve the result.

5. The Caller sends an Interest to the location provided by the Executor in the first transaction with a given Interest lifetime.
6. The Executor picks up the task from the pool and retrieves the parameters from the datastore. It responds to the pending Interest as soon as possible, or with new locations and time estimates if the results are not yet ready.

If this transaction is successful, then NSC completes. If there is a failure, we can safely repeat this transaction knowing that the only step missing is for the Caller to retrieve the result data.

4.6. NSC Naming Design

Naming design is normally application specific, but NSC proposes guidelines to simplify an implementation. NSC requires at least 2 parties and 3 namespaces:

- Executor
  - Function Name
  - Result Namespace (includes Delays)
- Client
  - Input Namespace

To simplify implementation, both Executor namespaces share a prefix. The Result Namespace also includes where delayed data is stored.

The next challenge is that each input and result need to be named with a unique identifier. The current design and implementation use a monotonically increasing number for both input and results. For high-scale systems though, that may be an issue once it reaches large number data names. So, an

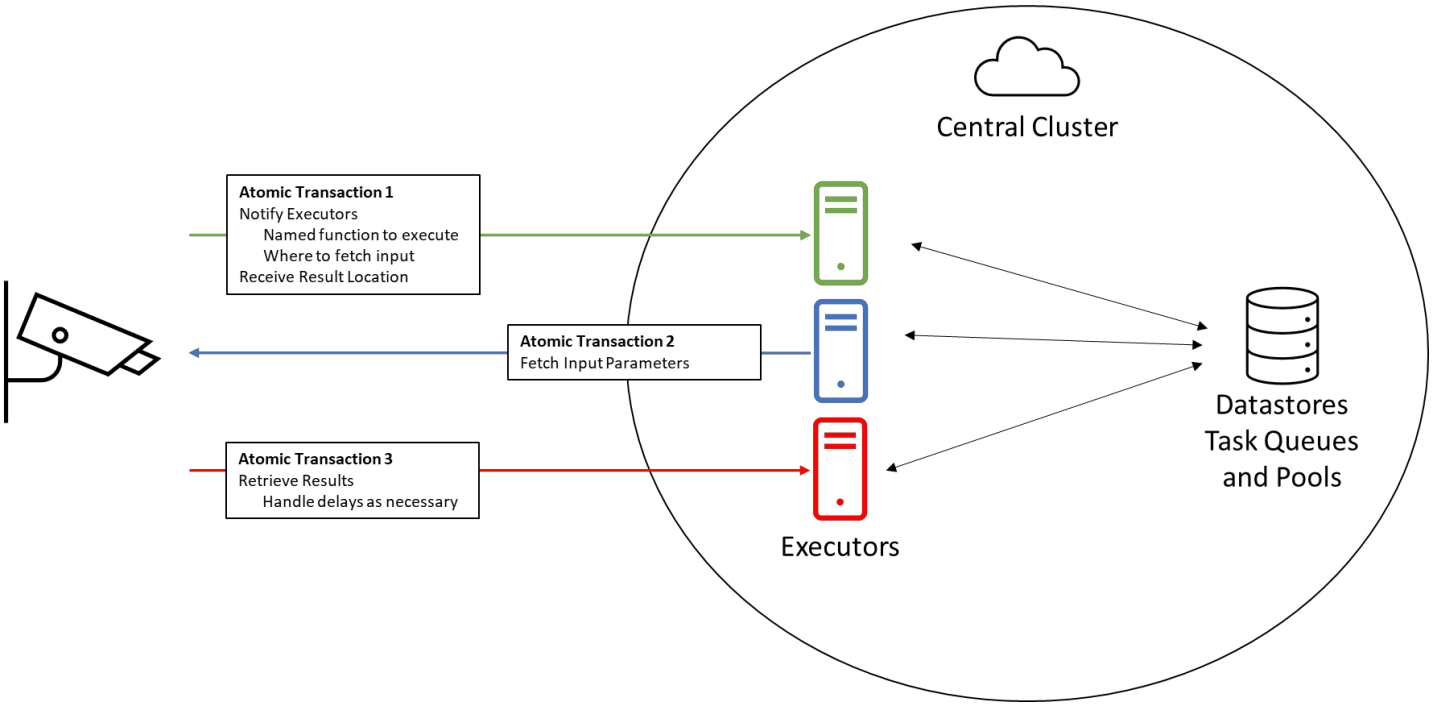

Figure 6: NSC Scalability Design

advanced implementation may breakdown the namespace by day, hour, or Client to limit name overlap and make it easier to generate a unique identifier.

So, the current design uses the following namespaces:

- Executor
  - /Executor/function
  - /Executor/function/results/1
- Caller
  - /Caller/function/inputs/1

5. Current Implementation

NSC was implemented using ndn-cxx [10] on an Ubuntu 18.04 Virtual Machine. A demo, featuring a credit-card checking function, shows NSC being used in a simple application.

The Caller emulates a Point-of-Sale (POS) device, which needs to validate credit-card numbers by sending them to an Executor. There is also an Executor which artificially adds a 10 second delay for all results, to simulate a long-running process.

The demo application makes the following simplifications:

1. The Caller and Executor identities and their trust anchor have been manually signed and bootstrapped on to the test device.
2. There are no Forwarding Hints used since each Caller and Executors knows how to navigate to the others namespace.
3. There is no timer estimation for results, so the Executor delays by a constant amount of 4 seconds each iteration.

All the implementation code can be accessed at: https://github.com/UCLAzlo/NDN_NSC

6. Future Goals

We believe NSC at this stage provides a framework for the most common remote code execution cases. However, there are many other use cases and NSC can be improved to meet other requirements.

6.1. Future Design Goals

The Signed Interest v0.3 spec will make the Interest Signature a separate TLV-Value, which means that it could be accessed and potentially stored to check against future transactions. That would mean that only the original Client could request their specific results, which enforces stronger privacy.

New authentication mechanisms, like EL PASSO, allows asynchronous and privacy-preserving authentication which might be useful for NSC as well [11]. That will help preserve the security of NSC while maintaining Client privacy, which is a desired feature in NDN applications.

For future performance improvements, it would be ideal if NSC had alternative communication patterns for smaller requests. If it is known that both the input parameter size and code execution time is small, then it would be preferable to have a single Interest-Data transaction which sends the input and retrieves the result at the same time. In select situations it would still be generally reliable and much more performant. This will require more measurements to detect when these performance improvements can be made reliably, and more communication infrastructure for how to fall back to the current NSC proposal when needed.

6.2. Future Implementation Goals

The current NSC implementation demo does not use encryption, and it would be beneficial for others if NSC already had encryption using NAC built into the example. Similarly, the current example assumes that identities and trust anchors are bootstrapped ahead of time, but it would be good to use some of the NDN configuration research for an automated enrollment example.

Finally, NSC would need to be converted into a standardized library to improve adoption. The current implementation provides a framework and design but takes significant work to implement into an application. Organizing NSC into a set of importable library functions would contribute more to the growing NDN codebase.

One way to accelerate NSC's usage is by fully implementing the example into the new NDN Python language. By having a CXX and Python use-case, NSC would attract a broader audience of application developers.

7. NDN Development – Lessons Learned

The demo implementation for NSC was our first experience developing for NDN with ndn-cxx. During the implementation, we learned several lessons.

The most valuable tools for developing NSC were the Client/Server examples included in ndn-cxx itself. The five lectures in 217B were also critical to become familiar with NDN development. Especially the lectures that showed the security model, and the general use of the ndn-cxx continuation passing style. Sharing the tempsensor/aircon/controller example on GitHub was another great reference for a simple NDN application. We believe that simple NDN applications like this should be formally documented and included with the built-in examples, as an aid to future application developers.

The first place we struggled, was with the Signed Interests. The Signed Interest spec on the NDN website described the latest spec version of Signed Interests v0.3, which includes the signature as a separate TLV-Value. However, we saw a different behavior in practice with our ndn-cxx application

[12]. Specifically, that the signature was being appended to the Interest name itself, which meant that we had to include the signature in the Data response name as well. This mismatch between the NDN documentation and ndn-cxx caused some confusion. Fortunately, help from another NDN researcher, Zhiyi Zhang, resolved this quickly.

The next struggle we had was using the ndn-cxx scheduler that comes from the boost asio event handler. The scheduler is required so the Server and Client respond to events asynchronously. There is an example of this in the ndn-cxx Server example, so we thought it would be simple to implement. However, we immediately had runtime crashes. We determined the cause was that our variables were declared in a different order. We were not familiar with the Boost library, and it did not seem obvious from the code or documentation that the declaration order of the Face and Scheduler should matter. Even when both objects are initialized later. We believe a small comment on the ndn-cxx example would be helpful to avoid this.

The last struggle was implementing multi-threaded code for results processing. This is partially due to our own lack of knowledge about the Boost Library that is used throughout ndn-cxx, but we ran into issues integrating Boost and C++ STL functions. Specifically, the Boost asio event handler was not compatible with STL asynchronous functions like promises. So, any multi-threaded code using the STL version of promises, would block instead of running in parallel. We implemented a fix by using the equivalent Boost function in all cases. Based on our experience, we do not think this requires separate documentation, but learning Boost is part of the learning curve for developing with ndn-cxx.

Our last comment is that the ndn-cxx documentation [13] was comprehensive and appreciated. The one area for improvement we can see is to provide more example uses for functions, like in other language libraries. We created many short programs to become more familiar with the proper usage of functions and type casting, and we believe others may have similar development questions.

8. Conclusion

Named Service Call, or NSC, provides a framework for Clients to execute code remotely on a given Server. NSC provides a security model to authenticate Clients, a method for sharing input parameter data and result data, and a dynamic system to allow Clients to correctly adjust for requests which take a long time to process. NSC's demo implementation proved that the idea is also technically feasible, even if there are still more steps required to make NSC a plugin type library for all applications.

NSC is theoretically slower than RICE, which created a handshake mechanism with passwords. And NSC is also slower than the secure DNMP framework which discovers new input data with an NDN Sync protocol. However, NSC is more generalized than both other frameworks, and works for a larger number of applications, and can be easy to adopt by new NDN developers.

9. References

[1] Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[2] gRPC. (n.d.). https://grpc.io/.

[3] GraphQL. (n.d.). https://graphql.org/.

[4] Rosenband, D. L. (1997, April 18). A Remote Procedure Call Library. http://web.mit.edu/6.033/1997/reports/dp1-danlief.html.

[5] Michał Król, Karim Habak, David Oran, Dirk Kutscher, and Ioannis Psaras. 2018. RICE: Remote Method Invocation in ICN. In ICN '18: 5th ACM Conference on Information-Centric Networking (ICN '18), September 21–23, 2018, Boston, MA, USA. ACM, New York, NY, USA, 11 pages. https://doi.org/10. 1145/3267955.3267956

[6] Kathleen Nichols. 2019. Lessons Learned Building a Secure Network Measurement Framework using Basic NDN. In 6th ACM Conference on InformationCentric Networking (ICN '19), September 24–26, 2019, Macao, China. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3357150.3357397

[7] Yingdi Yu, Alexander Afanasyev, David Clark, kc claffy, Van Jacobson, Lixia Zhang, Schematizing Trust in Named Data Networking. 2015 ACM

[8] Zhang, Z., Yu, Y., Ramani, S.j., Afanasyev, A., & Zhang, L. (2018). NAC: Automating access control via Named Data. In MILCOM 2018-2018 IEEE Military Com-munications Conference (MILCOM) (pp. 626–633).

[9] Yu Zhang, Zhongda Xia, Alexander Afanasyev, Lixia Zhang. A Note on Routing Scalability in Named Data Networking. 2019 ICC

[10] ndn-cxx https://github.com/named-data/ndn-cxx

[11] Zhang, Z., Król, M., Son-nino, A., Zhang, L., & Rivi\`ere, E. (2021). EL PASSO: Efficient and Lightweight Privacy-preserving Single Sign On. *Proceedings on Privacy Enhancing Technolo-gies, 2021(2), 70–87.*

[12] ndn signed interest documentation https://named-data.net/doc/ndn-cxx/current/specs/signed-interest.html

[13] ndn-cxx: NDN C++ Library with eXperimental eXtensions 0.7.0 documentation https://named-data.net/doc/ndn-cxx/0.7.0/doxygen/annotated.html