IPC Evolution thru Declarative Interface Generation

Position Paper

Nik Sultana
Illinois Tech
USA

Saket
University of Pennsylvania
USA

Andrew Zhao
University of Pennsylvania
USA

Shubhendra Pal Singhal
University of Pennsylvania
USA

Michael Kaplan
Peraton Labs
USA

Rajesh Krishnan
Peraton Labs
USA

Boon Thau Loo
University of Pennsylvania
USA

ABSTRACT

Inter-Process Communication (IPC) mechanisms are simple, OS-provided communication endpoints that do not typically accommodate program-level needs on latency, resource utilization, and mobility. But modern network-connected devices, particularly in IoT, have a wide variety of custom needs and frugal capabilities that standard networking stacks and programming interfaces do not cater for well. Thus IPC needs to evolve, but programmers would need to commit to new communication choices through their source code, which is difficult to change.

This position paper argues for the reimagining of IPC to benefit IoT through program-level tailoring of composable and reusable protocol building-blocks for computation- and data-management in distributed systems. We propose sprockets, a generalization of RPC beyond marshalling and synchronization. It incorporates programmer annotations about code semantics, program-level network-related functions, and performance expectations. This is a stepping stone towards the declarative synthesis of high-level IPC that better meets the program's communication needs.

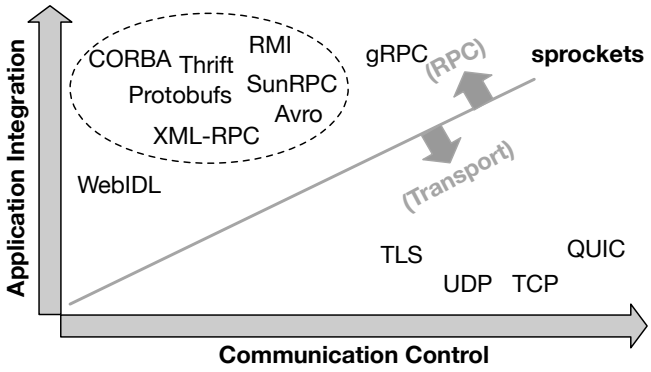

Figure 1: sprockets are a high-level IPC designed to serve the communication needs in IoT. It unifies features of RPC and transport protocols, so that "sprockets > RPC + IPC" (§3). Systems gathered in the dashed ellipse have significant overlaps in features. Program-level needs are addressed by RPC implementations, but these tend to be loosely coupled with transports. Transports afford more control on communication resources, such as error-correction, latency, timers, and buffer sizing, but typically do not differentiate between program data. By revisiting the line of abstraction between RPC and transports, we could search for an IPC interface that allows for easier offloading to hardware of program-level, communication-related needs such as caching and error handling.

1 INTRODUCTION

Traditional Inter-Process Communication (IPC) mechanisms—such as signals, shared memory, pipes, and sockets—present OS-supported services for the *communication of information*, but not for the *communication needs of programs* that use the IPC. IPCs typically present simple and fixed APIs for generality, usability and portability. It is then up to the programmer to wrap and maintain any program-specific communication behavior around these building blocks.

Communication is very important to modern programs, and essential in IoT. Programs tend to form part of distributed systems that involve both frugally-resourced IoT devices and more highly-resourced local or upstream servers that coordinate the devices and process their data.

The core problem with current IPC schemes is the limited *intent* that programmers can express. Programs have to "fight physics" [15] and IPC can be more helpful in mediating

program-level communication needs. The absence of communicating intent is problematic in two ways: **(i)** Most existing IPCs are too "low level": they are designed to provide a communication end-point, but do not capture *program-level* intent through knowledge of datatypes and function properties—such as idempotence, associativity, concurrency constraints, and other limitations on side-effects. We argue for the development of "high-level" IPCs that provide program-level abstractions to support modern use-cases. **(ii)** IPCs provide little *configurability*, which often takes the form of restricted namespaces (e.g., address and protocol families in the Berkeley sockets interface) or crude protocol customization through opaque calls to `ioctl()`. We argue for a compositional, property-guided approach that captures the program's latency and reliability preferences more intuitively, rather than through the rough customization of TCP's "one size fits all" approach, for example. In §2 we motivate this idea further.

IPCs evolve conservatively because of the legacy systems that rely on them, but there might be evolutionary pathways that do not disrupt legacy applications. Other parts of the communication ecosystem have evolved more readily, particularly in transport protocols such as QUIC [9], MPTCP [13], specialized datacenter transports such as NDP [6], and specialized protocols for IoT [2]. This evolution has come at the cost of interoperability with existing standards, since these schemes create fresh protocols, but applications are being modified in tandem of programmer's intent of exchange of data. sprockets provide a a flexible and featured concept which does not conflict with the existing system standards in addition. We argue that an improved IPC can work within and above existing standards by being transparent to them, by emulating techniques used in in-network computing and network middleboxes.

This slow evolution of IPC contributes to a growing **communication gap** between what programmers need and what IPCs can deliver. Current solutions to this problem fall into two camps: **(i)** more featureful Remote Procedure Call (RPC) interfaces or **(ii)** more configurable transports. Neither is ideal by itself, and combining them is not straightforward: they are separated by a layer of abstraction from one another as shown in Fig. 1, since current approaches tend to loosely-couple the two.

How IPCs could evolve. In this paper, we argue that new IPCs are needed to addresses this problem. These IPCs would be defined by *program-level* declarations. Such IPCs would offer better per-program customizability for the IPC's behavior and features. This would not only improve programmer convenience but also improve utilization of shared resources and improve security and performance. Current IPCs do not provide the right "knobs" to allow the programmer or some other control system to tune for this.

We propose an example IPC, called *sprockets*, that consists of the following: **(i)** High-level description of meta-properties of data-types and functions, such as side-effects and object sizes; **(ii)** High-level description of channel properties and program expectations, such as the types and quantities of loss that the program can tolerate, or whether a channel is simplex or duplex; **(iii)** High-level description of program-level behavior that the IPC should provide, such as caching (and its duration and cache size). **(iv)** The automatic synthesis of correct, stateful protocols that satisfy the high-level constraints **(i-iii)**. **(v)** A custom but stable API, and a performance overhead in which you only "pay for what you use". We envisage that sprockets would be library-provided, application-level mechanisms that expose partly- or wholly-offloadable functionality to specialized hardware.

sprockets do not *replace* existing IPCs, but rather build on them—in the same way high-level languages do not replace low-level languages. They are intended to provide a richer, reusable and mature front-end to stable communication interfaces. Further, they only target message-based IPC—not signals, for example.

There are two core ideas in sprockets' design: **(i)** it generalizes RPC to synthesize custom per-program protocols that are then executed through low-level (existing) IPCs; **(ii)** it is configurable both through the inclusion of custom processing code and through the combination of pre-packaged modules—for example, to provide bounded forms of reliability. These ideas enable IPC evolution through the extensible generation of program-level interfaces that combine OS-provided IPCs with PL-provided abstractions.

Have we been here before? The closest sprocket-style support that exists at the moment realize a very limited vision of the high-level, declarative IPC we are proposing. RPC and systems like protocol buffers [1] are focused on marshalling state based on type-level data descriptions, and are typically used in client/server peerings of applications. We envisage incorporating more information to the code synthesizer, such as explicit semantics of functions and function calls, program-level network-related functions, and performance expectations. Some of this information will be percolated down to the transport layer, to use a transport that satisfies constraints set by the programmer and provide a common entry point to both RPC and transports, as sketched in Fig. 1.

Core research questions related to declarative IPCs include:

- How do we express program-level communication intent?
- How do we express deployment-related performance expectations or constraints?

- How is communication intent translated into a protocol implementation, or a configuration of an existing protocol?
- How do we express and verify correctness criteria?

2 IPC EVOLUTION BENEFITS IOT

Programs' communication needs go beyond transferring bytes. Having a high-level IPC that can interpret more programmer intent would better address existing and changing trends in program communication: **(i)** Modern programs often run on heterogeneous platforms. Programmers need to manually bridge the gap between the barebones communications support provided by existing IPC. More adequate logic is needed to support modern application designs on modern platforms. For example, self-adjusting computation [3] is a general computation approach that can reduce resource use and improve performance, and could be coupled with an OS-provided IPC. **(ii)** *Cross-domain reasoning* (about what information can flow between programs) is left entirely to the programmer. Modern programs can make use of security-oriented architecture to mitigate against hardware vulnerabilities [8] by using trusted hardware and secure enclaves, but communication across domains is ad hoc. **(iii)** Communication patterns of distributed, heterogeneous programs are more complex, but IPCs currently provide no support for this despite commonly-occurring needs such as intermittent computation [7] for IoT or opportunistic networking on smartphones. **(iv)** Network-connected programs can avail themselves of different access networks, including WiFi or 5G [16]. Networks are increasingly offering rich communication and computation fabrics [14] for program-directed customization of usage, but existing IPCs are *too abstract* and leave it to programs to manage mobility [10] or timing constraints. This leads to mismatch in some settings, such as when communicating with autonomous vehicles [5]. **(v)** Future programs may need to distribute across more systems, such as in inter-cloud systems or edge clouds, thus strengthening the need for better communication support. **(vi)** Current IPCs present limited opportunity for application-oriented cross-layer optimization. While cross-layer optimization has been researched for specific applications and workloads [12], there does not exist a general facility for different applications, workloads, and criteria to use such optimization. Higher-level IPC can provide such a facility because of the logic they encapsulate. **(vii)** More sophisticated IPCs can define a larger *offload boundary* to utilize additional or specialized hardware, such as programmable switches [11] or NICs [4] that are already widely available.

3 AN EARLY PROTOTYPE: SPROCKETS

In this section we describe *sprockets*, an IPC design that realizes some of goals. sprockets have these features: **1) Generalize send/receive operations into custom communication operations.** Depending on what an application needs, functions can be synthesized to *both* send and receive a range of data over a set amount of time, and pass it through custom checks and transformations. Effectively, this creates a sychronization primitive that is customized for an application's needs. **2) Checkpointing of state adjacent to communication endpoints.** Transmitted state and metadata can be persisted for later replay, in case one of the peers crashes. The programmer is provided with hooks to either restore state from a checkpoint or to update a checkpoint based on persisted data after the last checkpoint. Old checkpoint information is purged. **3) Annotation of function metaproperties.** Functions are annotated to indicate whether they are side-effecting since this affects whether they can be called during a replay without affecting other state—to implement "at most once" semantics. Functions are also annotated to indicate whether they are idempotent, in which case calling the function more than once is a safe operation. This is done when the transport is not required to guard against repetitions. **4) Communication constraint annotation.** Various communication-related details need to expressed for the system to generate the right wrapper. These details include communication with a peer may be bidirectional, constraints on datagram size and the rate and latency of transmission. Hooks are provided in case any constraints are breached at runtime, so the program can adjust by signalling to the other end (if bidirectional communication is possible). Alternatively, fall-backs can be used—such as using different communication parameters or changing the IPC's communication behavior to send less data.

3.1 Protocol schema

Our initial design is based on the idea of a general *protocol schema* that can be instantiated to produce different types of protocols. The general idea is sketched in Fig. 2. Our early prototype (§3.2) allows us to pick-and-choose some features from TCP and provide program-level communication processing that goes beyond TCP.

This design combines with third-party systems, including serialization libraries and protocol transports, to provide better support for program-level communication needs. This support is chosen by the programmer based on a selectable set of features, including: largest data structure that can be transferred in a single unit (made to correspond to the underlying link's MTU), the communication's direction ($\mathbf{A} \rightarrow \mathbf{B}$, $\mathbf{B} \rightarrow \mathbf{A}$, or both, as shown in Fig. 2), the extent of reliability that is required (which indicates the required tolerance to

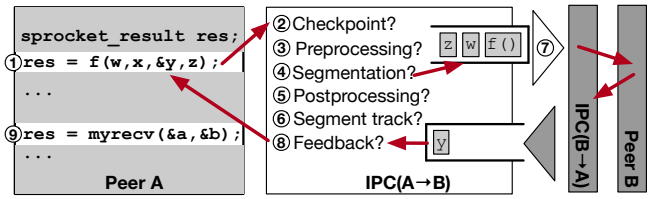

Figure 2: Processing stages in a sprocket IPC. ① A custom communication function is called. It generalizes both "read" and "write" operations (cf §3), and abstracts the custom-generated communication logic managed by sprockets. ② State can be persisted to mitigate against faults if the peered functions cannot ignore or recover from loss themselves. ③ A work-list of transformations is applied to memory references before they are serialized for communication. ④ Depending on underlying communication constraints, the interface decides what data to send and how to organize it into datagrams. ⑤ If required by the interface, additional functions are executed after segments are transmitted, for example to retransmit segments. ⑥ In parallel, and if required by the interface, segment delivery is tracked to ensure reliable transport. ⑦ At the other end, a complementary interface uses similar logic to deliver the data to the peer. ⑧ If required by the interface, response status and data is propagated back to the application synchronously or asynchronously. ⑨ Several ad hoc communication functions can be generated, each with their custom handling by the sprocket IPC.

loss and reordering of packets), and constraints on transmission rate and latency.

Program-level support is provided by selectable metafunctions such as caching, failure-detection and handling, and retaining a program-level log of events to bridge the gap between the program and the communication channel. By making some state explicit we can recover from failure through retries and checkpointing, as discussed earlier.

This support also relies on labelling the properties of code blocks and functions—such as idempotence, involution, and whether a function is side-effecting. This metadata is used to check whether the combination of constraints can be satisfied by the schema. This check is applied before generating a configuration that links together the invocation of sprockets with dependencies, such as OS-provided IPCs. Timers as well as checks related to information-flow control are provided by a monitor that is triggered by events; this overhead can be avoided if fewer runtime checks are needed.

Highlights of the implementation which are peculiar to our initial design would involve :

- Annotations provided for the program will undergo the first step of analysis where the program dependency graph(PDG) would initiate the data flow and control flow, making us aware, whether the program is eligible for the partition. Based on the function and data annotations, attached to individual enclave (referring to master/slave) and its dependency based on return values of the function. For instance : If a program just returns corresponding value would be converted as per the client/server's machine ISAs and declared as the only dependency which does not involve sharing any other local computation. If let's say it is a pointer, appropriate adjustments are made such that the memory correspondings between both the enclaves is maintained. Many such measurements based on whether the function is idempotent is also ensured during the process which is influenced by the annotations and the prospective data flow.
- Next step involves the verification of annotations made such that the division/partition made is in compliance with the programmer's perspective and in case of any conflicts, the annotation vs PDG flow is put on difference scale. This is followed by partitioning and generating the subsequent additional annotations(auto-generated) which would then allow both the enclaves to initiate and start their communication. One thing to note is that maintaining a difference in security level wrt to each program has been considered throughout the process.
- Such considerations directly influences the IoT sector as low-power requirements, meeting a deadline etc. plays a role which can easily be incorporated in the RPC. For instance, for the same program, if resending a packet(assuming idempotent) can bear one deadline miss, loss-delay resilience of our RPC can work that parameter while, if not, the features of annotation would ensure a safe execution where the loss-delay would no more be a part of program partitioned on enclaves.

3.2 Early evaluation

Our early prototype instantiates a protocol scheme as described in the previous section. An example instance is shown in Fig. 3 for a bidirectional exchange between two peers, either of which might fail at any moment, and whose communication can be disrupted. For our prototype we use encapsulate data to be transported over UDP. The peers are labelled "Caller" and "Callee", and the protocol precisely accounts for the state that needs to be maintained, failure conditions that might arise, and how they can be recovered from.

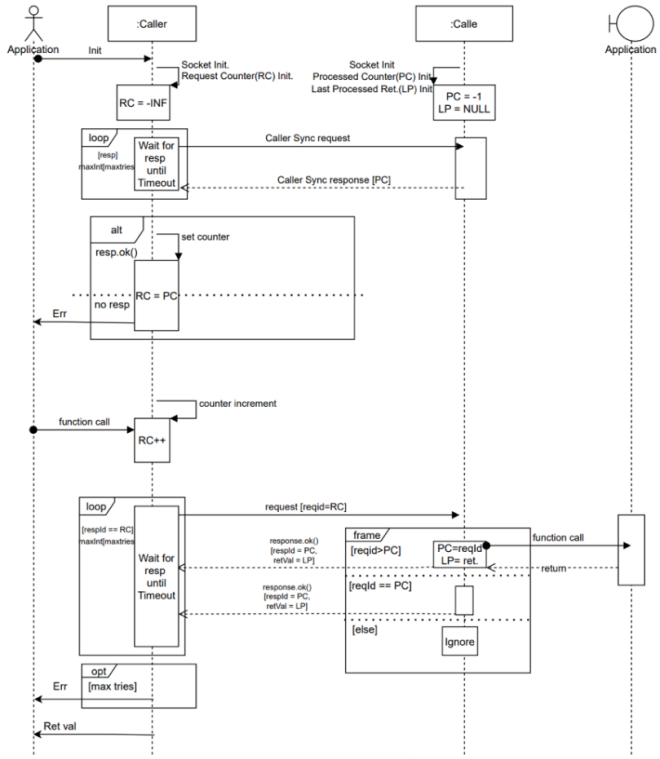

Figure 3: A lightweight protocol that provides a subset of TCP's behavior: in-order transmission of data. This version of the protocol had separate hand-shaking and transfer phases. This was later simplified into a single-phase protocol, and combined with call-side memoization.

Our current prototype only works for C, and does not automate the generation—it consists of a manually-invoked API that uses low-level IPCs. Realizing this vision further to automate this system across several programming languages could involve providing language-specific wrappers of our C code to begin with. We are continuing to expand the schema and prototype to support a broader range of features to support program-level communication needs. Additional features will be provided through interposition, partly inspired by in-network computation, and by leveraging features of transport protocols.

REFERENCES

[1] Protocol Buffers. https://developers.google.com/protocol-buffers/. Last accessed: January 2021.

[2] The Constrained Application Protocol (CoAP). https://tools.ietf.org/html/rfc7252. Last accessed: January 2021.

[3] Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative Self-Adjusting Computation. *SIGPLAN Not.*, 43(1):309–322, January 2008.

[4] Adrian Caulfield, Paolo Costa, and Manya Ghobadi. Beyond SmartNICs: Towards a Fully Programmable Cloud. In *IEEE International Conference on High Performance Switching and Routing*, June 2018.

[5] János Czentye, János Dóka, Árpád Nagy, László Toka, Balázs Sonkoly, and Róbert Szabó. Controlling Drones from 5G Networks. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, SIGCOMM '18, page 120–122, New York, NY, USA, 2018. Association for Computing Machinery.

[6] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 29–42, New York, NY, USA, 2017. Association for Computing Machinery.

[7] Matthew Hicks. Clank: Architectural Support for Intermittent Computation. *SIGARCH Comput. Archit. News*, 45(2):228–240, June 2017.

[8] Matthew Hicks, Cynthia Sturton, Samuel T. King, and Jonathan M. Smith. SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs. *SIGPLAN Not.*, 50(4):517–529, March 2015.

[9] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 183–196, New York, NY, USA, 2017. Association for Computing Machinery.

[10] Yuanjie Li, Qianru Li, Zhehui Zhang, Ghufran Baig, Lili Qiu, and Songwu Lu. Beyond 5G: Reliable Extreme Mobility Management. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 344–358, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 143–157, Boston, MA, February 2019. USENIX Association.

[12] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network Stack Specialization for Performance. *SIGCOMM Comput. Commun. Rev.*, 44(4):175–186, August 2014.

[13] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving Datacenter Performance and Robustness with Multipath TCP. *SIGCOMM Comput. Commun. Rev.*, 41(4):266–277, August 2011.

[14] Ahmad Rostami, Peter Öhlén, Mateus Augusto Silva Santos, and Allan Vidal. Multi-Domain Orchestration across RAN and Transport for 5G. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 613–614, New York, NY, USA, 2016. Association for Computing Machinery.

[15] Jonathan M. Smith. Fighting Physics: A Tough Battle. *Commun. ACM*, 52(7):60–65, July 2009.

[16] Dongzhu Xu, Anfu Zhou, Xinyu Zhang, Guixian Wang, Xi Liu, Congkai An, Yiming Shi, Liang Liu, and Huadong Ma. Understanding Operational 5G: A First Measurement Study on Its Coverage, Performance and Energy Consumption. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 479–494, New York, NY, USA, 2020. Association for Computing Machinery.