Transparently Securing Kafka, Istio-style, with up to 300% Higher Performance than Native TLS in Microservice Environments

Jayanth Gummaraju

BanyanOps Inc.
jayanth@banyanops.com

Kafka is emerging as the epicenter of today's microservice deployments with ephemeral clients running in containers using Kafka for fast, asynchronous communication. Securing and auditing Kafka in these dynamic environments with high performance can be very challenging, especially at scale. In this article, we propose a new approach that not only provides transparent, scalable, end-to-end security between producers and consumers, but also gives a significant boost to mTLS performance for data-in-transit without any changes to code/config.

1. Introduction

Kafka is emerging as a dominant messaging platform in microservice environments. Whereas one-to-one, request/response-style communications are usually handled using direct, synchronous, service-to-service communication (e.g., REST, grpc), one-to-many/many-to-one asynchronous communications are better handled using a Pub/Sub messaging platform like Kafka. Today's typical microservice deployments involve ephemeral Kafka clients (producers and consumers) running in dynamic container orchestration environments such as Kubernetes, Marathon, or Docker Swarm. Kafka servers usually serve as an external data service that its clients use for asynchronously communicating data (e.g., SMACK stack).

Securing Kafka in such microservice environments is very challenging -- it involves not only controlling access to sensitive data-at-rest in Kafka, but also securing and auditing data flowing between short-lived producers/consumers through Kafka servers. Kafka natively provides basic security features such as Java SSL, Kerberos/SASL, and simple ACLs. However, as we move into dynamic microservice environments with multiple tenants and clusters, native mechanisms can be challenging to operationalize and inadequate for compliance in regulated environments. Organizations are on their own to ensure end-to-end encryption between producers and consumers, enable multi-factor authentication, segment access to Kafka, setup secure key management (e.g., PKI), handle weak/leaked credentials including revoking access to compromised entities, setup fine-grained role- and time- based access controls, meet ever-changing audit and compliance requirements, and ensure developers spanning multiple teams are accessing Kafka following security best practices.

To address these challenges, we introduce a new approach using sidecars (aka micro-engines) that secures Kafka transparently, without needing any changes to Kafka clients, servers, or the underlying platform. Our approach has also been advocated by the recently announced open service mesh platform, Istio, from Google/IBM/Lyft albeit in the context of Kubernetes services. Whereas sidecars are primarily deployed in container-based green-field environments from the ground-up for operational needs like load balancing, this article focuses on deploying them in both container and non-container (process) based brown-field environments for security, and tailored to Kafka setups. Sidecars (can be process or container) are deployed alongside Kafka components including producers, consumers, brokers, and zookeeper servers. The sidecars authenticate each component using multiple factors (e.g., service account, metadata, etc.) and assign them cryptographic identities, intercept all connections and transparently upgrade them to

mTLS, and exchange identities for fine-grained topic-level time-based access controls. In addition, to guarantee end-to-end security between producers and consumers, data-at-rest is also encrypted and can only be accessed by authorized consumers. All of this is accomplished without changing a single line of code or config in any Kafka component.

In this article, we demonstrate how such an approach can be leveraged for protecting Kafka to provide superior security, strong authNZ, higher TLS performance, and several operational benefits compared to using just the native Kafka security. In particular, this approach provides:

- State-of-the-art **secure transparent encryption** between all Kafka components independent of Java versions on the servers and language limitations on the clients
- **End-to-end producer-to-consumer encryption and access control** including when data is at rest
- **Segmentation** between Kafka components and/or other applications without punching firewall holes
- **Multifactor identity and strong authentication** using decorated X.509 certs for clients, brokers, and zookeeper
- **Higher than native mTLS performance** made possible by using high-speed TLS libraries, ciphersuites and other optimizations
- **Secure, simplified PKI infrastructure** using short-lived certs (rotated every few mins), secure key management (e.g., keys not exposed on disk), and secure bootstrapping
- **Transparent encryption between zookeeper servers** (zookeeper has no native support yet)
- **Independent audit** of accesses and policy revisions for fast-changing producers/consumers that are hard to track
- **Fine grained access controls** such as leased access to clients for only approved topics, during specific times, etc. using role/attribute-based access controls (RBAC/ABAC)
- **Deep visibility** (e.g., topic, consumer group, etc.) into network traffic while preserving end-to-end encryption between client and server
- **Separation of application development from security considerations**, allowing developers to just focus on application logic and velocity
- **Homogeneous controls** across multiple services, not limited to just Kafka

Our evaluation results show that for typical microservice deployments, where the number of concurrent connections is high (>=64) and the record sizes are small (<= 1KB) our system provides huge (~200-300%) performance improvement over native TLS implementation both in terms of Throughput and Response Times using the latest supported Java version (1.8.x) in Kafka. Surprisingly, even with just one connection, we saw up to ~300% throughput improvement, making the native implementation a broader problem than just in microservice environments. The performance benefits are primarily due to using high-performance Banyan sidecars written in Go and limitations in native Java SSLEngine compared to Go crypto/tls (details in Section 4). We expect to get similar performance results (perhaps better?) by extending other high-performance sidecars like Envoy, which is written in C++ and uses OpenSSL/BoringSSL TLS library. Looking ahead, on a preliminary port of Kafka to the pre-released Java 1.9 (1.9+181) the performance gap is narrowed, but still substantial (up to 36%). Although there is CPU cost associated with using a sidecar (~15% for 32MB/s and 64 concurrent connections), the security, performance, and operational benefits provided by this approach easily outweigh the CPU overhead for most deployments.

2. Overall Architecture

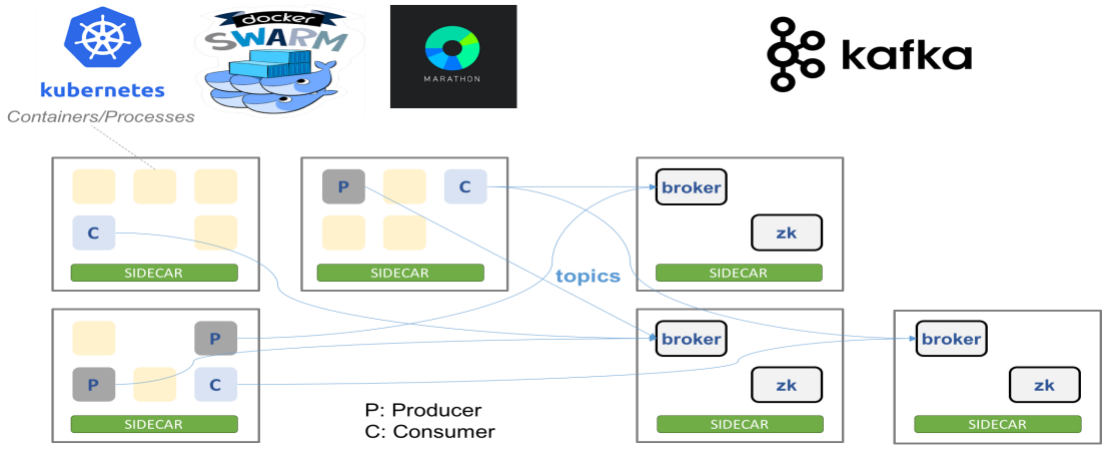

Figure 1: Typical Microservices Environment with Banyan's shared sidecars for Kafka security

Figure 1 shows a typical microservice environment accessing Kafka. On the left we have producers, consumers and other workloads running in container-based clusters such as Kubernetes, Swarm, or Marathon. On the right we have Kafka brokers and zookeeper instances that hold data and metadata shared by Kafka clients, viz. producers and consumers. On every host, Banyan's shared sidecar (can be a process or a container) is deployed that intercepts all communications between Kafka clients and brokers, between brokers and zookeeper, between zookeeper servers and between brokers. There is no change needed either in the code or config of clients, brokers or zookeeper. In fact, none of them are even aware that the communication is getting intercepted and/or modified in any way.

The overall flow of data from producers to consumers using sidecars is as follows. Producer creates a topic (a set of partitions aka message queues) and sends messages to the Kafka broker (leader) responsible for the topic partition. The shared sidecar running on the producer VM intercepts the connection to the leader, and depending on the policy coordinates with the sidecar on the leader to upgrade the connection to use TLS. If mutual auth is also requested, the producer and broker are authenticated by the sidecars using one or more factors such as their service accounts, labels, container images/SHA of binary, etc. Each sidecar validates the remote side's identity based on the certificates exchanged between the sidecars as part of the TLS handshake. The sidecars are also responsible to verify if the policy allows the producer to send data to this topic at this time.

If everything looks good, the sidecar forwards either the decrypted data after TLS, or re-encrypts the messages (payload) with a topic-specific AES-256 symmetric key if end-to-end encryption is requested before sending it to the leader. The leader receives the messages, persists them to disk, and sends replicas of the messages to other brokers (followers). Depending on the policy, sidecars in the broker VMs can again be used to upgrade their connection to use mTLS.

Just like producer to broker data-flow, broker to consumer data-flow can be upgraded to use mTLS, and access to specific topics can be controlled, according to the policy specification. Consumers (and consumer groups) subscribe to specific topics to read messages from the partition leaders, also store an offset as a marker of how much they have consumed. If the policy requires mTLS, when a consumer connects to the broker, the sidecar on the consumer VM intercepts the connection, validates the consumer, and establishes TLS connection to the broker-

side sidecar after validating the broker identity during handshake. The broker-side sidecar, in turn, validates the broker identity and consumer identity (mTLS handshake), checks for READ access to the topic and then sends the consume request to the broker. The data returned by the broker is either sent as is, or decrypted using topic-specific AES-256 symmetric key before sending on the TLS channel.

Zookeeper is the distributed consensus store responsible for electing a controller, managing broker membership, and configuring topics along with quotas and ACLs. Zookeeper is just another service and any communication between brokers and zookeeper, and between zookeeper servers themselves can similarly be encrypted and controlled for access using the sidecars.

The TLS certificates needed for encrypting and authenticating connections is handled transparently and securely using a high-performance PKI infrastructure. Once the sidecars are securely bootstrapped, they are responsible for issuing certificate requests on behalf of the producers and consumers to Banyan Certificate Authority, which is optionally integrated with the CA (certificate authority) and HSM (hardware security module). These certificates are decorated with additional metadata about producers and consumers using fields like SNI, OU, etc. to add authentication factors and enable easy auditing. Furthermore, the certificates/keys for the clients and brokers are rotated frequently (e.g., once/hour) and only stored in memory for better security.

For a more detailed description of Kafka operation itself, please refer to the Kafka documentation.

3. Security and Operational Benefits

Using a sidecar model provides several benefits to both the Security and Operations teams in a variety of environments. In this section, we discuss these benefits specifically in the context of Kafka running in dynamic, microservice environments.

Simplified operations

- **Separate application logic from security requirements**: It's hard to follow security best practices (e.g., no credentials on disk, short-lived certs) in multi-tenant environments, where producers/consumers are written by developers spanning multiple teams, in multiple languages (polyglot environments), without compromising developer velocity. By using an approach that is completely transparent to the application, it becomes very easy to deploy a highly secure solution even in brown field environments, and in the Kafka case, with potentially better performance (next Section).
- **Visibility into ephemeral Kafka clients**: Microservice environments have short-lived producers/consumers that could be running for just a few seconds, from any number of hosts. Getting full visibility into these interactions is essential for smooth operations.
- **Single pane of glass**: The same dashboard used for managing Kafka can be used for other services in container environments like Kubernetes, Mesos, Docker Swarm, and data services like MySQL, Redis, etc.

Dynamic security

- **High performance, end-to-end secure encryption**: Sidecars enable using the latest, greatest ciphersuites, independent of the Java versions in Kafka components and end-to-end encryption between producers and consumers. Java SSL has been historically known

to be slow to upgrade to the most secure and high-performance ciphers, and has its own set of vulnerabilities different from other SSL engines like OpenSSL. For example, SHA1 TLS certificates were deprecated only in July 2017 (8u_141 release), although SHA1 has known security issues since at least 2014. Furthermore, Kafka clients are becoming increasingly polyglot and ensuring all these languages have the best security implementation that are compatible with Java SSL is a non-trivial problem.

- **Multi-factor mutual Auth and secure PKI**: Sidecars, after securely bootstrapping, authenticate Kafka components (clients, brokers, zookeeper) using multiple factors such as service accounts, labels, binary SHA, location, etc. Once the identity is established, the sidecars obtain and rotate short-lived certificates/keys on behalf of the components and do not store them, addressing common challenges such as revocation, stolen keys and credential leakage (e.g., no truststore password in configs, long-lived certificates, etc.). Banyan PKI infrastructure is also easily integrated with existing organization-wide CA infrastructure and HSMs that store root CA keys.
- **Fine-grained access control**: Sidecars enable access control, where it matters, when it matters, and at the desired granularity. For example, for less trusted consumers, the operators can easily provide leased access to specific topics for a short duration (e.g., 1 hr). Such policies can be specified using the well-known RBAC/ABAC frameworks and used in conjunction with native Kafka policies for additional security.

Compliance and Audit

- **Audit unauthorized accesses**: Get detailed information about clients accessing Kafka in real-time and historically: e.g., container/process name and id, host, container image/binary. Additionally, this data can be easily sent to SIEMs such as Splunk for further analysis and forensics.
- **Segment for compliance (e.g., PCI-DSS) without punching firewall holes**: Kafka clients and brokers can be segmented using their multi-factor cryptographic identities rather than punching firewall holes using their IP address and ports. This approach to segmentation, called Identity-based segmentation, works well even in dynamic environments where IP addresses are changing/unreliable and also helps thwart attacks and avoid misconfigurations related to IP address or application spoofing.
- **TLS everywhere for compliance**: There are several compliance requirements for data-in-transit -- from storing keys safely, ensuring privacy of sensitive data, to using the latest ciphersuites (e.g., NIST 800-52 and FIPS 140-2 guidelines for HIPAA, PCI-DSS 3.2 for credit card data). Accommodating this growing set of strict guidelines is easily achieved using a separate security layer like transparent sidecars, which are independent of the application and can be easily upgraded and tested.

4. Performance Implications

In this section, we discuss the experiments we performed to understand the overhead associated with enabling TLS between Kafka clients and brokers in microservice environments. We describe our setup, performance results and analyze the tradeoffs of using sidecars vs. native TLS in the rest of the section.

Experimental Setup

Broker/Server	Azure DS-12 v2 (4 vCPUs, 28GB memory)
Client	Azure DS4 v2 (8 vCPUs, 28GB memory)

Oracle JDK	Java builds (1.8.0_131-b11, 1.8.0_144-b01)
Versions	Kafka: 2.10-0.10.0.1; Banyan: 0.4.x
OS	Ubuntu 14.04 linux-4.4.0-78-generic
Ciphersuite	TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
Benchmarks	rdkafka_performance, kafka-producer-perf-test++

Table 1: Experimental Setup

Table 1 gives the experimental setup we used for our performance studies. We ran several experiments using standard Kafka performance tools with default settings: rdkafka_perf from librdkafka and kafka-producer-perf-test from Apache Kafka distribution. We also extended kafka-producer-perf-test so that we could see the effect of increasing number of connections. We applied Kafka-3554 patch that allows varying number of threads for producers. The main change we made was to use a new connection for each thread instead of reusing the same connection for all threads. We also added code to ignore the initial jitter in performance due to a large number of concurrent connections initiated in the beginning. The code is available on github.

A typical Kafka production deployment in a microservice environment entails tens of Kafka brokers and hundreds of Kafka clients accessing thousands of topic partitions with varying record (message) sizes. But in order to focus on just the overhead of using TLS and avoid any performance impact due to other factors (e.g., replication, latency/throughput between brokers, etc.), we decided to use one broker, one client, and one topic partition, but vary the attributes that matter most in microservice environments: number of connections, record sizes, and applied load. We verified our results were similar with multiple brokers -- each new broker adds a new TLS connection, scaling TLS overhead linearly. To emulate the impact of many clients we increased the number of connections in kafka-producer-perf-test, as described previously. We measured throughput, response times, and CPU overhead in both the client and broker machines, and ran experiments multiple times and for long enough duration to get statistically significant results. That said, additional experiments can be performed to characterize the results more precisely (e.g., tail latency, different hardware/software versions, micro-benchmarks, etc.), which is beyond the scope of this article.

Performance results

Overall, our results show that for typical microservice deployments, where the number of concurrent connections is high (>=64) and the record sizes are small (<= 1KB) our system provides ~2-3X performance improvement over native TLS implementation both in terms of Throughput and Response Times when using recent JDK 1.8.x builds (last couple of months). The performance benefits are primarily due to using a high-speed TLS library (Go 1.8.x crypto/tls) vs. native Java SSLEngine and fast ciphersuite (AES128-GCM).

Our results are conservative and would be even better if we did not use persistent connections between the client and kafka broker. Sidecars optimize non-persistent connections using techniques such as tunneling multiple TCP connections into one connection between sidecars, and reusing TLS sessions with tickets. These optimizations mainly help in reducing the overhead of setting up TCP and TLS connections. Note that TCP handshake incurs a round-trip cost (SYN/SYN-ACK/ACK), and TLS handshake involves multiple round-trips and an asymmetric key exchange, which is very expensive compared to symmetric encryption used in steady state.

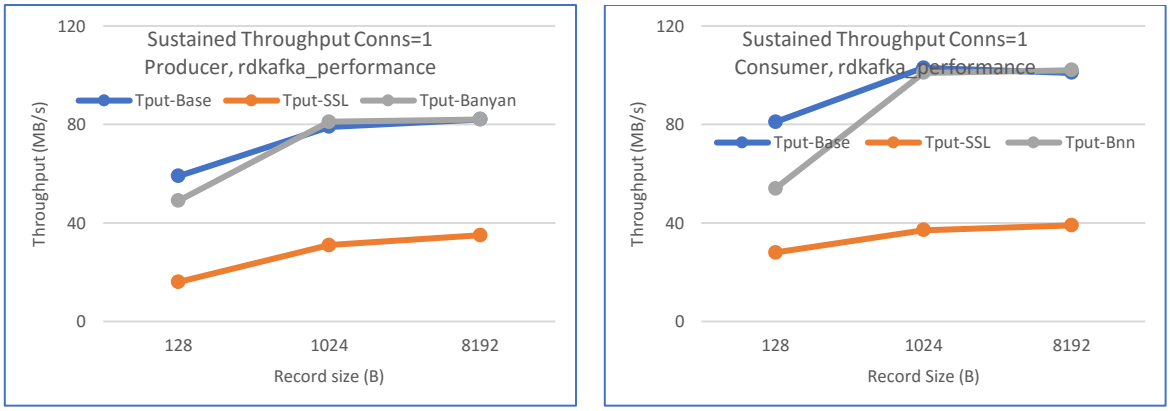

Figure 2: Sustained throughput as a function of record size

Figure 2 shows sustained throughput as a function of record size. For small record sizes (128B) the overhead of native TLS (Tput-SSL) compared to using sidecars (Tput-Bnn) is 3.06X (producer) and 1.92X (consumer), and compared to not using TLS is 3.68X (producer) and 3.67X (consumer). Sidecars themselves incur an overhead of 1.2X (producer) and 1.5X (consumer) compared to not using TLS. The insecure, non-TLS case (Tput-Base) has the additional benefit of doing zero copy transfers using sendfile for consumers. As the record size increases (1K, 8K), the performance overhead of TLS using sidecars is negligible compared to no TLS. The native TLS implementation, however still has a huge performance impact compared to using sidecars: 2.34X - 2.61X (producer) and 2.61X-2.75X (consumer).

Note that the throughput tapers off at ~80MB/s for producer tests. This is because we get limited by the disk bandwidth and we're running with the default configuration (acks=1) which writes to log before acknowledging the client. For faster disks, we expect the curves to be higher, but the relative throughputs between no-TLS and TLS (native, bnn) shouldn't change much. If network were the bottleneck, factors such as TLS record overhead need to be considered. However, for long-lived connections used in our experiments, the TLS record size quickly reaches 16KB as the TCP window grows. Hence, the TLS record overhead (~40B) becomes negligible.

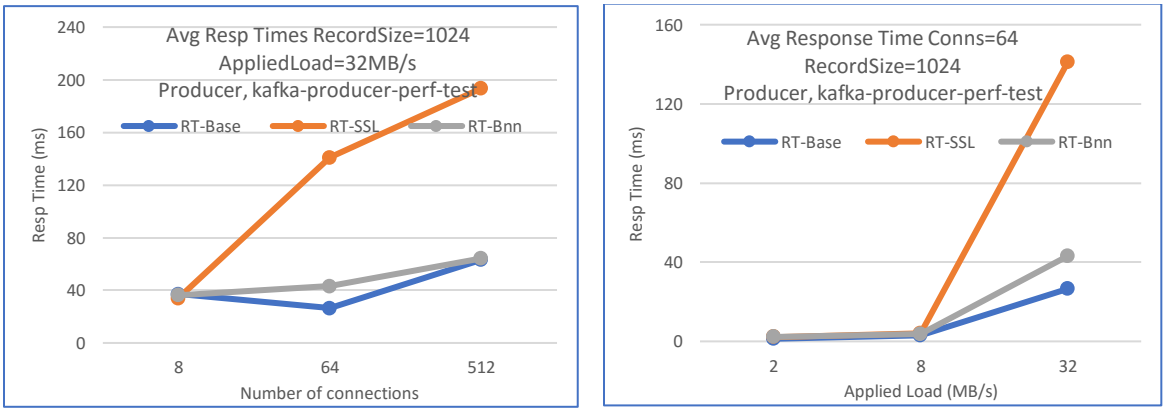

Figure 3: Avg response time as a function of number of connections and applied load (lower is better)

Figure 3 shows the average response time as a function of record size and applied load for a producer test. The results show that for small applied loads and small number of connections, there is not much difference in performance between no-TLS (RT-Base) and TLS with both native (RT-SSL) and sidecar (RT-Bnn) implementations. However, as the applied load increases (32MB/s) and the number of concurrent connections increases (64, 512), whereas the response

times of no-TLS and TLS-with-sidecars become comparable, the performance differential between native TLS and TLS-with-sidecars increases to more than 3X (up to 3.25X).

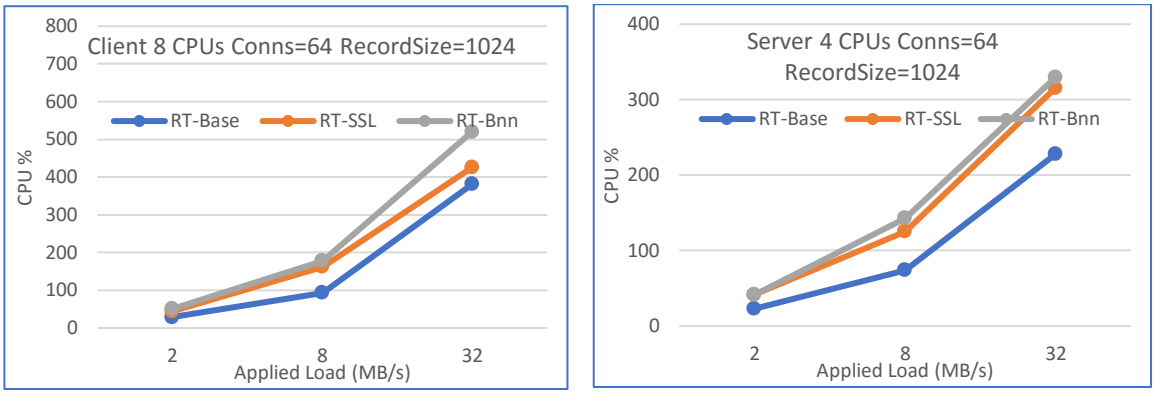

Figure 4: CPU overhead on client and broker machines (lower is better)

Figure 4 shows the CPU overhead associated with using TLS and more importantly using sidecars. The results show that CPU overhead with native TLS is ~15-20% for small applied loads (2MB/s), but quickly rises to ~45% on the client and ~90% for large applied loads (32MB/s) on the server. The overhead on server matters more because on the client this overhead is amortized over 64 connections (< 1% per connection) and typically clients are spread across multiple machines. Using a sidecar incurs additional CPU overhead of 1.6% per connection (95% across all clients) and ~15% on the server compared to using native TLS.

Performance Analysis

Kafka has known performance issues with TLS primarily due to reliance on Java native SSL (KAFKA-2561, Librdkafka-920). The performance numbers discussed in these issues are similar or worse than those reported in this article. This section explores the performance tradeoffs of using sidecars rather than Java native SSL for TLS encryption.

The performance difference between using native Java SSL vs. sidecars is primarily affected by two factors:

- Using sidecars can entail extra copies: If TLS is implemented in user-space (like in our experiments), data intended for Kafka first enters the kernel's networking stack, then is processed and copied to the sidecar in user space where it gets decrypted. The data is then copied back into the kernel and then copied again into the user space of Kafka broker. This increases latency and incurs CPU overhead. However, modern CPUs leverage REP STOSD and SSE/SIMD instructions very efficiently while performing `memcpy`, which significantly reduces the CPU overhead and latency associated with copies. Moreover, for highly latency sensitive and high bandwidth applications, TLS can be implemented in the kernel (e.g., new Linux (4.13) or kernel module) to reduce the overhead even further.
- Java SSL is slow: The JDK SSLEngine has known performance overheads – very poor GCM performance, unable to effectively utilize all the cores of the CPU, garbage collection overheads, and limitations in some of its APIs (e.g., SSLEngine.unwrap()) causing synchronization issues. The performance would be much worse than reported here with older versions of Java because prior to version 1.7u40 doesn't even have support for AES-NI instructions, which is essential for reasonable TLS performance. Trying to fix this issue by using a different SSL library (e.g., Netty) in Kafka is non-

trivial, creates its own set of maintenance challenges, and would not address client-side issues for other languages (e.g., PHP doesn't reuse connections across requests).

Switching to CBC mode from GCM can reduce the performance overhead, but it comes at the expense of weaker encryption and is not recommended in secure environments. Whereas both CBC and GCM modes use XORing plaintext and data, CBC XORs with data that the attacker knows (IV or ciphertext from the previous block), GCM XORs plaintext with a "nonce" generated using a counter sent through a block cipher and also doesn't need padding, which makes it more secure. For example, CBC mode is vulnerable to timing-based side channel attacks such as Lucky Thirteen (CVE-2013-0169) and padding oracle attacks (CVE-2016-2107). Therefore, CBC ciphers have been declared obsolete (e.g., Google chrome only supports AEAD ciphers) and does not meet compliance requirements (e.g, NIST 800-52 for HIPAA).

In order to verify that the overhead is indeed a result of poor GCM implementation, we ran two experiments: using the weaker CBC instead of GCM for Java 1.8.x, and using a preliminary port for the pre-release Java 1.9+181 (to be released end of Sep 2017) instead of Java 1.8.x. Java 1.9 has recently added support for `pclmulqdq` x86-64 instruction that should help with optimizing GHASH used in GCM. Figure 5 shows the Throughput as a function of varying record sizes. Using CBC instead of GCM, reduces the TLS overhead large record sizes (1.15X vs. 2.38X using GCM for 8KB), but for small record sizes, the overheads are still very significant (2.45X for 128B and 1.62X for 1KB). As expected, using Java 1.9 (with GCM) is much closer in performance to the sidecar implementation, but is still slower for small record sizes (1.36X for 128B) likely due to other SSLEngine limitations, as discussed above. Furthermore, upgrading to the soon-to-be-released Java 1.9 to reduce TLS overhead involves upgrades to not just Kafka server but also Java clients potentially being used by multiple teams across the organization and with big changes in Java 9, it is likely going to be a slow process.

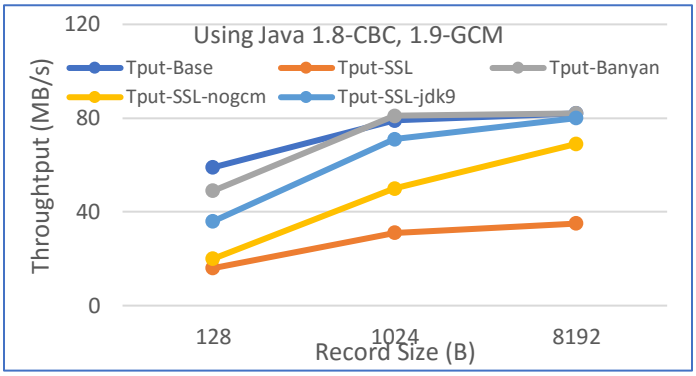

Figure 5: Throughput using other Java configurations

5. Discussion

Leveraging Istio-style transparent sidecars in microservice environments using Kafka can provide several operational and security benefits. Although not useful as a load-balancer in these environments (which is the common function associated with sidecars), this architecture enables an application independent layer that can simplify operations management, provide dynamic security, and streamline compliance and audit without impeding developer velocity. Being completely transparent to the application and underlying platform allows our approach to not be limited to just Kafka, but easily extends to other services and frameworks (e.g., MySQL, Redis, etc.) in both green-field and brown-field deployments.

In addition, this approach can provide better than native performance for security functions like mTLS by leveraging state-of-the-art libraries, ciphers, and other TLS optimizations for Kafka servers and the polyglot clients, independent of the language limitations. Although using sidecars does incur additional CPU overhead due to copies, the high-speed `memcpy` implementations in modern CPUs and the relatively cheap cost of cores makes it a non-issue in most deployments. These performance results are not specific to Kafka and extend to other frameworks that use Java native SSL. For example, in addition to traditional frameworks like WebLogic and Hadoop, some of the newer ones (e.g., SMACK stack including Spark and Cassandra) are likely subject to such performance degradation for TLS encryption.

X.509 certificates enable short-lived credentials, additional authentication factors, and fine-grained access controls like time-based leased access. However, solely relying on X.509 certificates for authentication may not be sufficient in some environments – e.g., if a client application is acting on behalf of multiple end-users and needs to provide different views depending on the identity of the end-user (e.g., WebLogic). We are working on extending our approach to support such multi end-user environments.

With TLS 1.3 around the corner that would ban ciphers with vulnerabilities at any level, and frequently updated compliance regulations, the sidecar approach allows easy upgrades to existing infrastructure to support latest, greatest security practices without changing a single line of application code or config.

Acknowledgements

We'd like to thank Rean Griffith, Eli Collins, Jay Kreps, Srinivas Mantripragada, Mendel Rosenblum, Carl Waldspurger, Alberto Begliomini, Zack Butcher, and Wencheng Lu for their insightful comments on the early drafts of this article.

