Wykorzystanie eBPFw celu optymalizacji pracy aplikacji — PoC

Źródła:

- https://www.usenix.org/conference/nsdi23/presentation/zhou
- https://www.usenix.org/conference/nsdi24/presentation/zhou-yang
- https://conferences.sigcomm.org/sigcomm/2023/workshop-ebpf.html

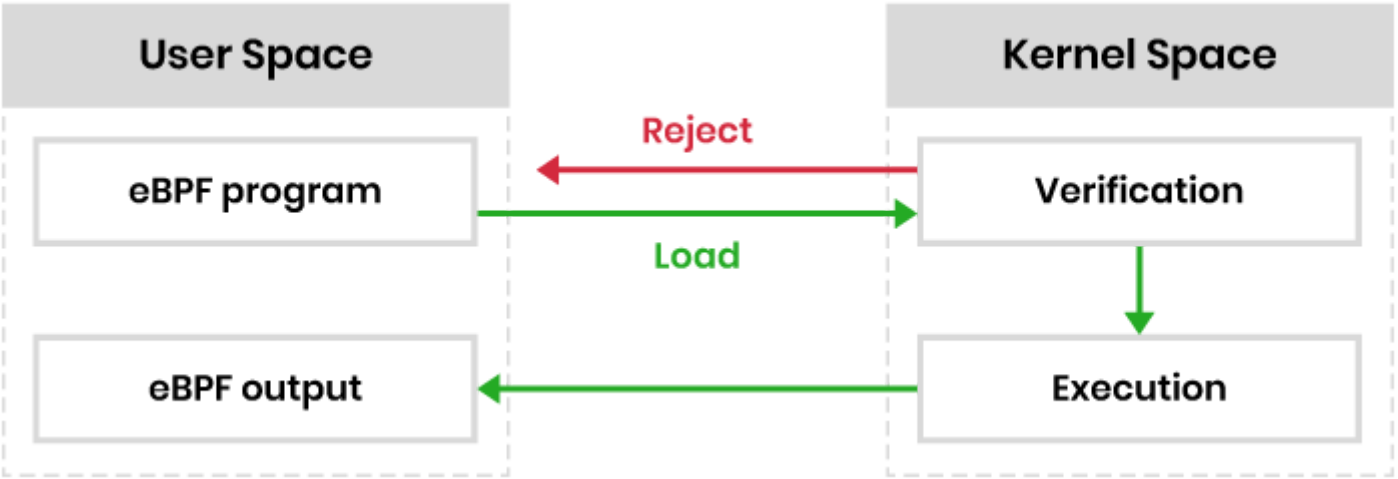
SIGCOMM

Extended Berkeley Packet Filter (eBPF) - technologia jądra (w pełni dostępna od Linuksa 4.4), umożliwia uruchamianie programów (bytcode Berkeley Packet Filter (BPF), który wykorzystuje określone zasoby jądra) bez konieczności dodawania dodatkowych modułów lub modyfikowania kodu źródłowego jądra. Można ją sobie wyobrazić jako lekką, piaskownicową VM w jądrze Linuksa (chociaż piaskownica nie wie, co uruchamia, a dla eBPF mamy weryfikację)

Przypadki użycia eBPF:

- **bezpieczeństwo**: rozszerzenie możliwości wyświetlania i interpretowania wszystkich wywołań systemowych oraz zapewnianie widoków wszystkich operacji sieciowych na poziomie pakietów i socketów (połączenie kontroli i widoczności wszystkich aspektów)
- **sieci**: połączenie wydajności (kompilator JIT — efektywne niemal jak natywnie skompilowany kod w jądrze) i programowalności (dodawanie parserów protokołów, modyfikacja logiki przekazywania) sprawia, że jest to dobry kandydat dla wszystkich wymagań przetwarzania pakietów
- **śledzenie i profilowanie**: można dołączania programy eBPF do punktów sondowania jądra i aplikacji użytkownika oraz mieć wgląd w zachowanie aplikacji w czasie wykonywania, jak również systemu.
- **monitoring**: zamiast polegać na miernikach i statycznych licznikach udostępnianych przez system operacyjny, eBPF pozwala na generowanie visibility events oraz gromadzenie i agregację niestandardowych metryk w jądrze (szeroki zakres źródeł)

Architektura eBPF:



W fazie *verification* sprawdzane jest, czy program eBPF się nie zapętli i przez to nie zablokuje jądra. Ponadto eBPF musi spełnić kilka dodatkowych checków (rozmiar, stan rejestrów, skoki poza granice) co odróżnia go od LKM (Linux Loadable Kernel Modules).

Po spełnieniu checków program eBPF jest ładowany i kompilowany do jądra, a następnie (po otrzymaniu sygnału) ładowany do codepath, gdzie bytecode wykonuje swoje instrukcje po inicjalizacji.

	Execution model	User defined	Compilation	Security	Failure mode	Resource access
User	task	yes	any	user based	abort	syscall, fault
Kernel	task	no	static	none (code reviews)	panic	direct
eBPF	event	yes	JIT, CO-RE	verifier, JIT	error message	restricted helpers, kfuncs

Podobnym konceptem jest P4, ale bierze na celownik bardziej sieciowe architektury (eBPF procesory ogólnego przeznaczenia).

Dodatkowo eBPF jest mocno powiązany z aplikacjami (control plane) i środowiskiem (data plane). Rust ma na celu ulepszenie komponentów jądra i koncentruje się na zastąpieniu kodu, który często był mniej kontrolowany (sterowniki urządzeń).

Sieci

W jaki sposób eBPF czyni kernel platformą dla cloud native, która można komponować?

- przyspieszenie rozwoju oprogramowania infry cloud native (skrócenie pętli feedbacku między developerem a użytkownikiem (łatanie jądra))
- użycie cech jądra dla danego case'a (bezpieczeństwo na metadanych Kubernetesa)
- przeniesienie przetwarzania danych bliżej źródła
- szybszy feedback à propos researchu
- oddziela zmianę zachowania jądra od jądra bazowego
- reużywalność bloków z jądra (FIB)
- głęboki, nisko-nakładowy wgląd w jądro
- ma bezpieczniejszą wersję języka C

XDP — użytecznie dla sieci (obsługa wielu buforów o MTU > 4k, podpowiedzi do odciążania NIC), ale:

- BPF jest uruchamiany dla każdego pakietu zamiast wsadowo.
- W niektórych przypadkach dane dostępne w BPF mogą nie znajdować się jeszcze w cache'u procesora.
- Wsparcie dla dołączania wielu programów w warstwie XDP wciąż na TODO
- Niektóre sterowniki wymagają rekonfiguracji kanału ethtool RX/TX przed użyciem.

tcx — sterownik generyczny, działający na wejściu po GRO i wyjściu przed GSO

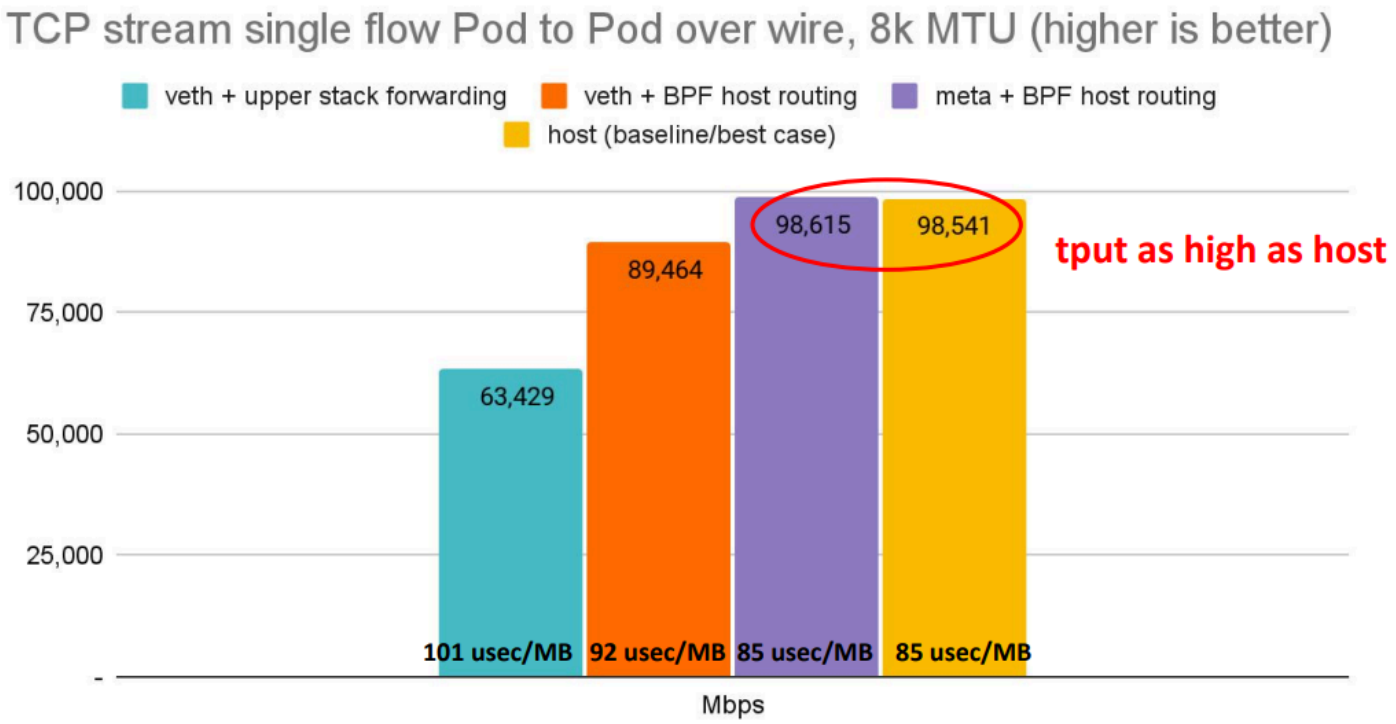
- Oparty na skb, a zatem bogaty w funkcje, elastyczny i szybki
- Idealny dla sieci kontenerowych lub dowolnego rodzaju ruchu, który ostatecznie kończy się w lokalnym gnieździe (lub z niego pochodzi) - uzupełniając XDP
- EDT do kształtowania przepustowości, pełny dostęp do fib jądra, NAT46x64, ...
- Może współdziałać i wymieniać informacje z programami XDP

cgroup socket hooks — działa na podstawie struct sock_addr / gniazdo jako dane wejściowe

- brak dostępnego skb
- przydatne do implementacji proxy lub L4LB dla ruchu klastrowego wschód/zachód (np. Cilium)
- uzupełnia XDP dla ruchu klastrowego przenosząc przetwarzanie bliżej źródła zdarzenia.
- Przenosi NAT na pakiet do NAT w czasie połączenia

sk_lookup hook — BPF może wybrać gniazdo inne niż 5-krotność skb.

- Zakres dołączania jest zgodny z przestrzenią nazw sieci
- Przekierowanie na gniazdo nasłuchujące w przypadku TCP, dowolne gniazdo dla UDP
- Przypadkiem użycia jest przezwyciężenie ograniczeń bind(2) API, np. powiązanie gniazda z podsiecią (x.y.z.0/24, port 80) i kierowanie pakietów przeznaczonych dla IP na dowolnym porcie (a.b.c.d/32, dowolny port) do pojedynczego gniazda



Kontrola przeciążeń

W kontekście kontroli przeciążeń, w kernelu hosta można ustawić parametry przetwarzania pakietów (początkowe okno CWND), ale nie jest to przyjemne (moduły kernela lub zmiana Linuxa xD). I tutaj z pomocą przychodzi nam eBPF:

Rozszerzenie procesu decyzyjnego TCP o wcześniejsze doświadczenia związane z przepływami zebrane i udostępnione wśród aktywnych przepływów przy użyciu modułów eBPF.

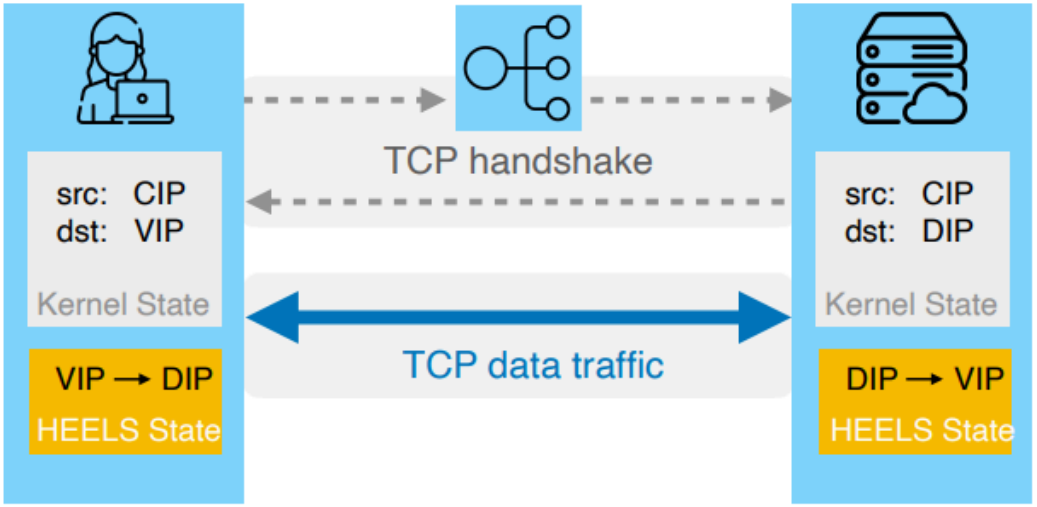
1. Wcześniejsze doświadczenie w przepływie jest zagregowane
2. eBPF przechowuje/udostępnia stany przepływu (dla widoku ścieżki danych)
3. Haki eBPF wywołują różne reakcje TCP/IP, jednocześnie uzyskując dostęp do współdzielonych informacji o przepływie.

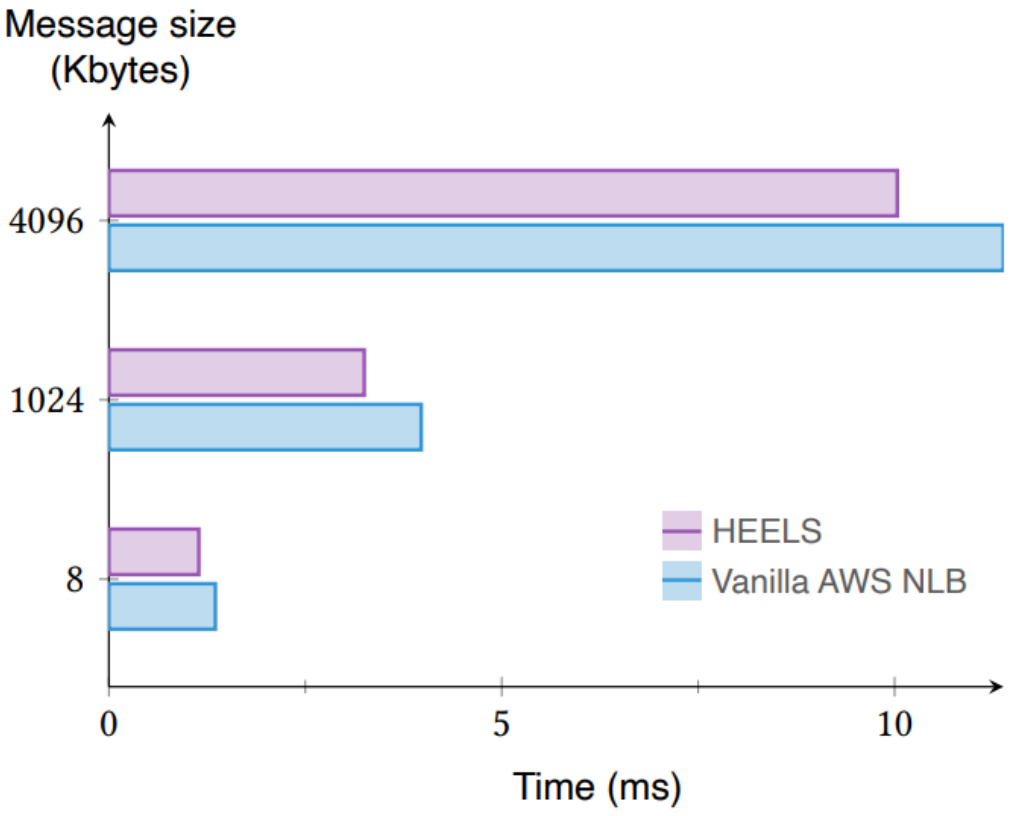
Load balancing

CRAB — skalowalne i wydajne rozkładanie obciążenia;

- wymaga niestandardowego load balancera (niekompatybilny z rzeczywistymi load balancerami)
- wymaga zmian w jądrze po stronie klienta poprzez bezpośrednie łatanie jądra lub ładowanie modułów
- ale jest słaby wdrożeniowo

I tutaj cały na biało wchodzi HEELS (Host-Enabled eBPF-Based Load Balancing Scheme)





Nsdi

Niedawne przesunięcie wąskiego gardła z pamięci masowej do sieci ze względu na postępy w dziedzinie pamięci DRAM z zasilaniem bateryjnym i szybkiej pamięci NVRAM. Zagłębia się w wyzwania związane z wdrażaniem szybkich rozproszonych transakcji w pamięci przy użyciu technik sieciowych omijających jądro, takich jak RDMA i DPDK, podkreślając ich kompromisy w zakresie bezpieczeństwa, izolacji i wydajności. Zaproponowano wykorzystanie stosu sieciowego jądra z przetwarzaniem pakietów sterowanym przerwaniami zamiast technik omijania jądra, wprowadzając eBPF jako kluczowy czynnik umożliwiający.

Mapy eBPF to wbudowane w jądro struktury danych używane do przechowywania stanów programów eBPF. Mogą one zawierać do 2^32 - 1 elementów, każdy o maksymalnym rozmiarze 2^32 - 1 bajtów. Mapy te są statyczne i muszą być zadeklarowane ze stałym rozmiarem. Występują w różnych typach, takich jak tablice, tablice per-CPU, stosy i kolejki, oferując funkcje wyszukiwania i aktualizacji. Jedną z ich kluczowych cech jest możliwość współdzielenia między różnymi programami eBPF i procesami przestrzeni użytkownika, ułatwiając komunikację i koordynację w całym systemie.

Programowanie eBPF ma ograniczenia wynikające z weryfikacji jądra. Brakuje w nim obsługi dynamicznej alokacji pamięci, a pętle muszą być określane statycznie. eBPF nie obsługuje również wysokopoziomowej synchronizacji wątków, takiej jak Mutex, ze względu na potencjalne ryzyko uśpienia jądra. Obsługuje jedynie blokadę spinlock, ale ze ścisłymi ograniczeniami, takimi jak brak możliwości wywoływania funkcji podczas utrzymywania blokady.

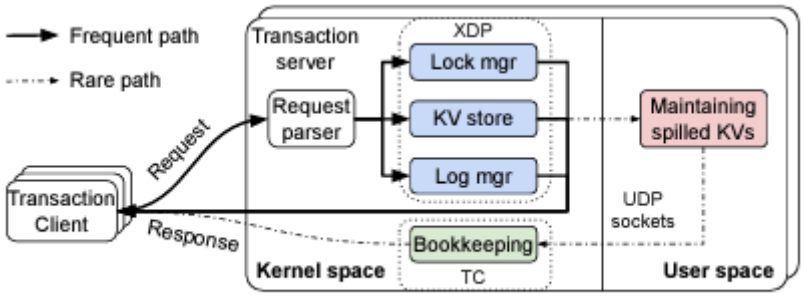


Figure 2: DINT's high-level architecture.

DINT optymalizuje przetwarzanie transakcji poprzez odciążenie jądra od operacji na częstych ścieżkach, zmniejszając narzut stosu jądra. Serwery transakcji przechowują większość stanów w pamięci jądra za pomocą map eBPF i obsługują żądania w jądrze za pośrednictwem programu eBPF dołączonego do hook'a XDP. Aby wysłać odpowiedź, DINT modyfikuje ładunek pakietu żądania, zmieniając jego przeznaczenie do przenoszenia wiadomości odpowiedzi z powrotem do klientów.

DINT używa map tablic eBPF do tworzenia statycznych tablic stanów blokady w przestrzeni jądra. Każdy identyfikator blokady jest powiązany ze współdzielonym stanem blokady za pomocą funkcji skrótu. Upraszcza to operacje blokowania, ale nieznacznie zwiększa ryzyko niepowodzenia w pozyskiwaniu. Blokady mogą wystąpić, jeśli klient próbuje uzyskać dwie blokady mapujące na ten sam stan (zakładając wyłączne blokowanie).

Magazyn klucz-wartość DINT zarządza mapowaniem między kluczami i wartościami, obsługując operacje takie jak GET, INSERT, UPDATE i DELETE. W przeciwieństwie do konwencjonalnych magazynów wartości klucza w przestrzeni użytkownika, które wykorzystują dynamiczne alokacje pamięci, które nie są odpowiednie dla eBPF, DINT radzi sobie z tym wyzwaniem, przechowując wartości klucza w asocjacyjnej pamięci podręcznej w jądrze, wspieranej przez mapę eBPF o stałym rozmiarze. Taka konfiguracja zapewnia efektywne wykorzystanie pamięci i ułatwia szybkie wyszukiwanie. Gdy wiadro jądra zawiera zbyt wiele wartości klucza, DINT przenosi niektóre z nich do przestrzeni użytkownika, utrzymując wydajność poprzez równoważenie obciążenia. Każde pole wartości ma stały rozmiar, aby pomieścić większość obiektów transakcji, zapewniając kompatybilność z różnymi obciążeniami.

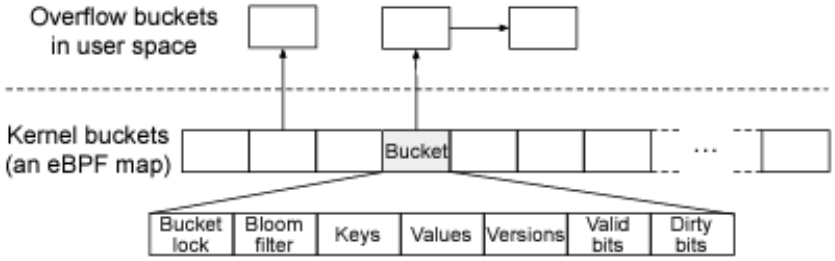
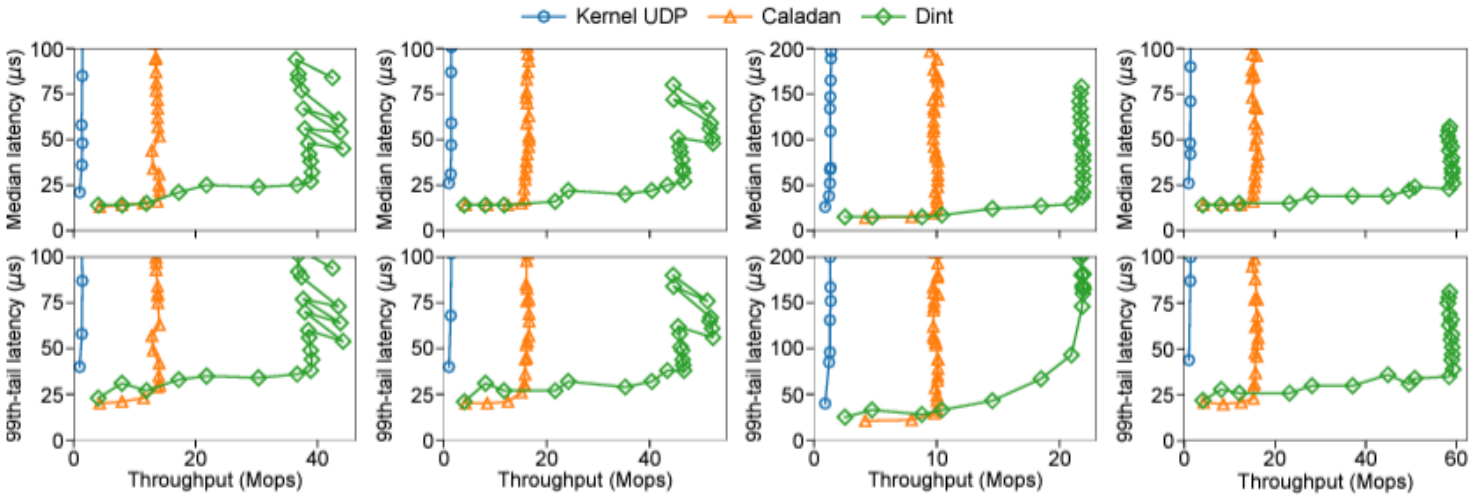


Figure 3: The layout of the key-value store in DINT (assume using the version-based locking).

Wysokowydajne rozproszone systemy transakcyjne przechowują dzienniki transakcji w pamięci w celu odzyskiwania danych po awarii. W miarę wzrostu dzienników systemy obcinają je lub przenoszą na dyski, gdy przestrzeń dziennika zostanie przekroczona. DINT wybiera obcinanie i dąży do szybkiego rejestrowania w eBPF dla normalnej pracy, jednocześnie wspierając złożone odzyskiwanie offline w przypadku awarii.

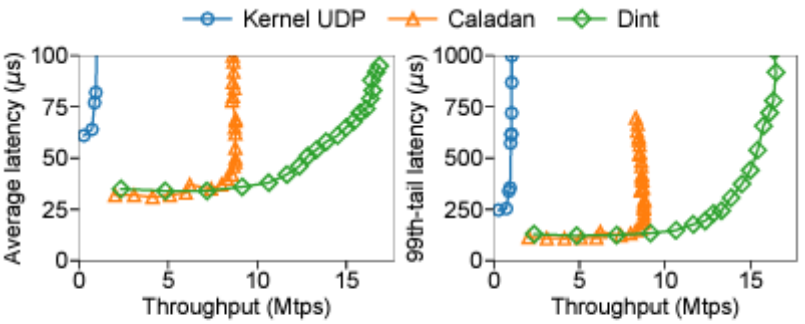
Wykresy 5a i 5b ilustrują, jak opóźnienia (zarówno mediana, jak i 99th-tail) menedżerów blokad 2PL i OCC różnią się w zależności od osiągniętej przepustowości dla różnych systemów. Każdy system działa podobnie w przypadku dwóch menedżerów blokady, przy czym menedżer blokady OCC jest nieco szybszy ze względu na brak operacji atomowych dla odczytów wersji w OCC. Ogólnie rzecz biorąc, DINT osiąga 3,1×-3,2× wyższą przepustowość niż Caladan, z 0%-8%/5%-55% wyższą medianą/99th-tail opóźnieniem bez obciążenia, podczas gdy jądro UDP działa znacznie gorzej niż inne. DINT doświadcza jednak wahań przepustowości przy dużych obciążeniach.



(a) 2PL lock manager. **(b)** OCC lock manager. **(c)** Key-value store. **(d)** Log manager.

Figure 5: Microbenchmark load-latency curves (both median and 99th-tail).

Wykresy 6a i 6b przedstawiają średnie i 99th-tail transakcji różnych systemów w zależności od przepustowości. DINT osiąga 1,9× wyższą przepustowość transakcji niż Caladan, przy 6%-10%/12%-16% wyższym średnim/99th-tail opóźnieniu bez obciążenia. Wyższa przepustowość DINT wynika z bezpośredniej manipulacji i przekazywania nieprzetworzonych pakietów ethernet/UDP natychmiast po ich otrzymaniu przez sterownik NIC, w przeciwieństwie do Caladana, który polega na wysokopoziomowej abstrakcji zorientowanej na połączenie. Dodatkowo batching skutecznie łagodzi narzuty związane z obsługą przerwań w DINT, utrzymując wysokie obciążenia na serwerach transakcyjnych. Jednak to grupowanie prowadzi do większych opóźnień w porównaniu z Caladanem opartym na odpytywaniu z pominięciem jądra, przy czym DINT ma o 3 µs/14 µs wyższe minimalne średnie/99th-tail opóźnienie.



(a) Average latency vs. tput. **(b)** 99th-tail latency vs. tput.

Figure 6: OCC on TATP workload. Mtps = Million transactions per second.

CPU usage

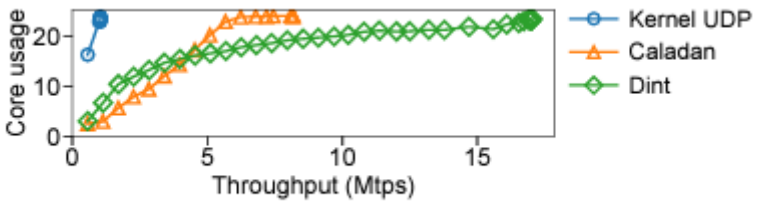
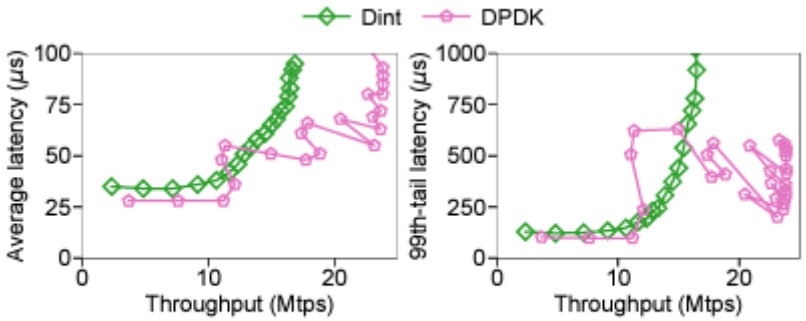


Figure 8: Core usage vs. throughput (on TATP).



(a) Average latency vs. tput. **(b)** 99th-tail latency vs. tput.

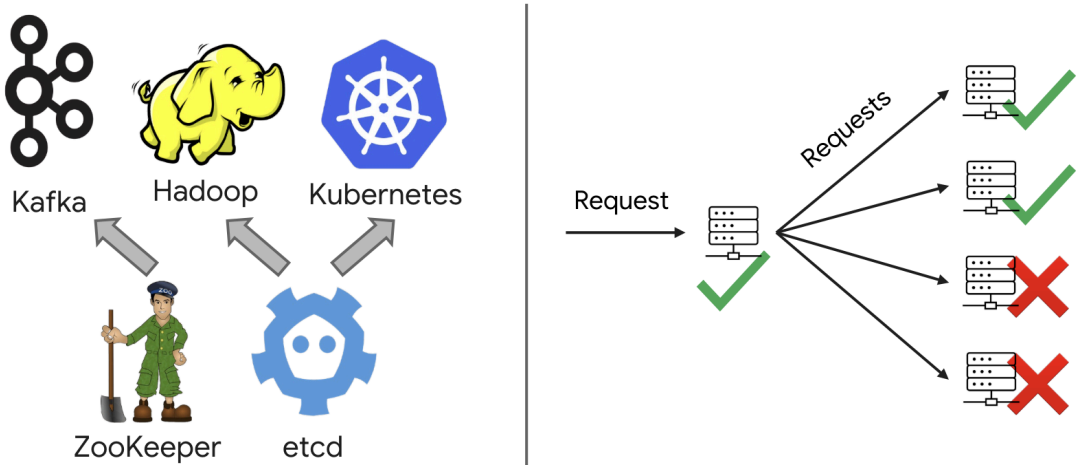
Figure 9: Comparing raw DPDK with DINT (on TATP).

W tym kontekście „surowe DPDK” odnosi się do busy pooling'u partii pakietów transakcyjnych (do 64, podobnie jak NAPI jądra) z karty sieciowej, przetwarzania żądań transakcji, a następnie bezpośredniej modyfikacji i przekazywania pakietów z powrotem w partii jako odpowiedzi. W związku z tym jest bardziej wydajny niż linia bazowa Caladan, ale wymaga busy pooling'u wszystkich rdzeni. Ogólnie rzecz biorąc, DINT osiąga 71% surowej wydajności DPDK przy 21%-25%/24%-28% wyższym średnim/99-tym opóźnieniu bez obciążenia.

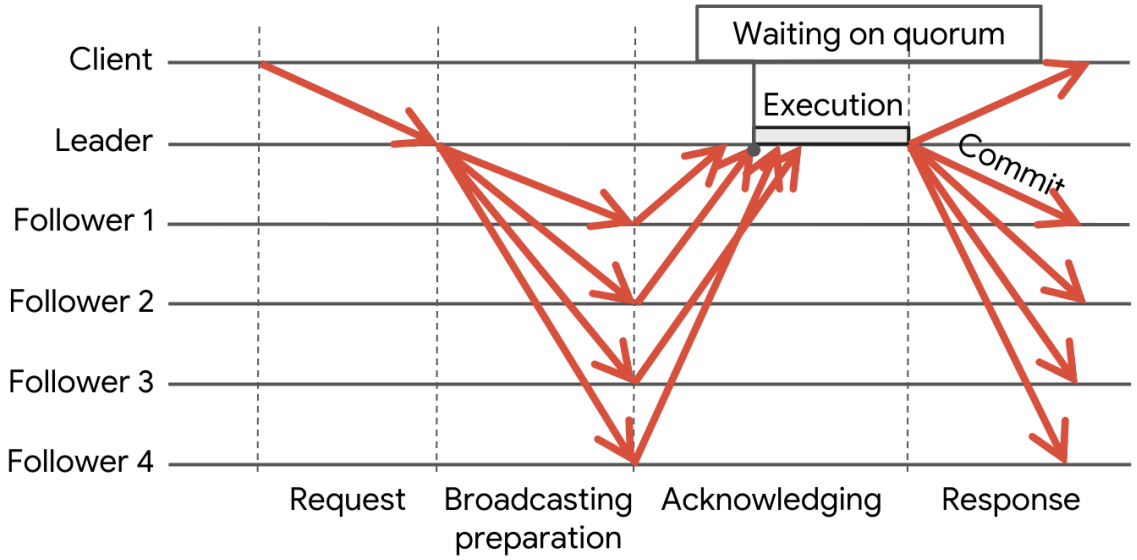
Accelerating Distributed Protocols with eBPF

eBPF ma kluczowe znaczenie w optymalizacji rozproszonych protokołów, takich jak Multi-Paxos, dzięki możliwości wykonywania operacji bezpośrednio w jądrze systemu operacyjnego, co znacznie redukuje potrzebę częstych zmian trybu między przestrzenią użytkownika a przestrzenią jądra. Rozproszone protokoły charakteryzują się znacznym ruchem sieciowym, co przekłada się na częstą zmianę trybu. Optymalizacje przedstawione w artykule są uruchamiane w jądrze przed stosem sieciowym zapewniając tę samą funkcjonalność jak byłyby zaimplementowane w przestrzeni użytkownika, jednocześnie unikając narzutu związanego przez zmianę trybu. Autorom artykułu poprzez zastosowanie eBPF w protokole Multi-Paxos udało się polepszyć przepustowość o 128.4% i opóźnienie o 41.7%.

Cloud applications need consensus protocols for high availability

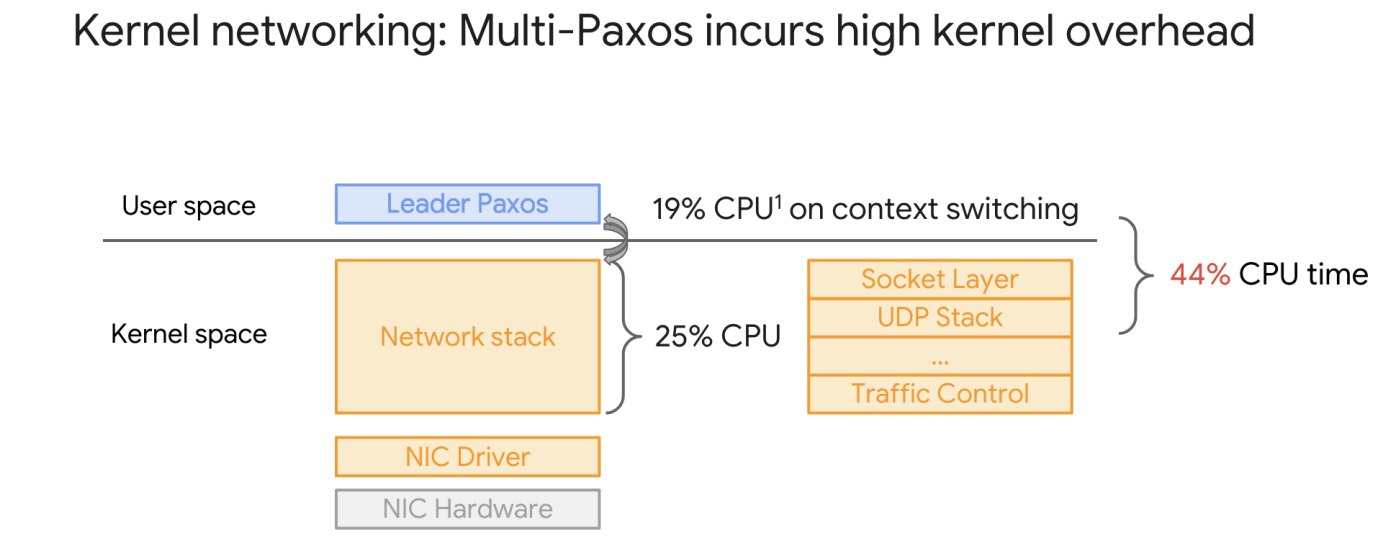


W typowej implementacji protokołu Multi-Paxos z pięcioma replikami lider musi wysłać i otrzymać łącznie czternaście komunikatów w ramach przetwarzania jednego żądania. Te komunikaty to między innymi wiadomości przygotowawcze (preparation), potwierdzenia (acknowledgment) i zatwierdzenia (commit). Proces ten jest wysoce obciążający dla stosu sieciowego jądra ze względu na wielokrotne przejścia między przestrzenią użytkownika a jądrem.



In this example, the leader node invokes networking APIs **14** times per request

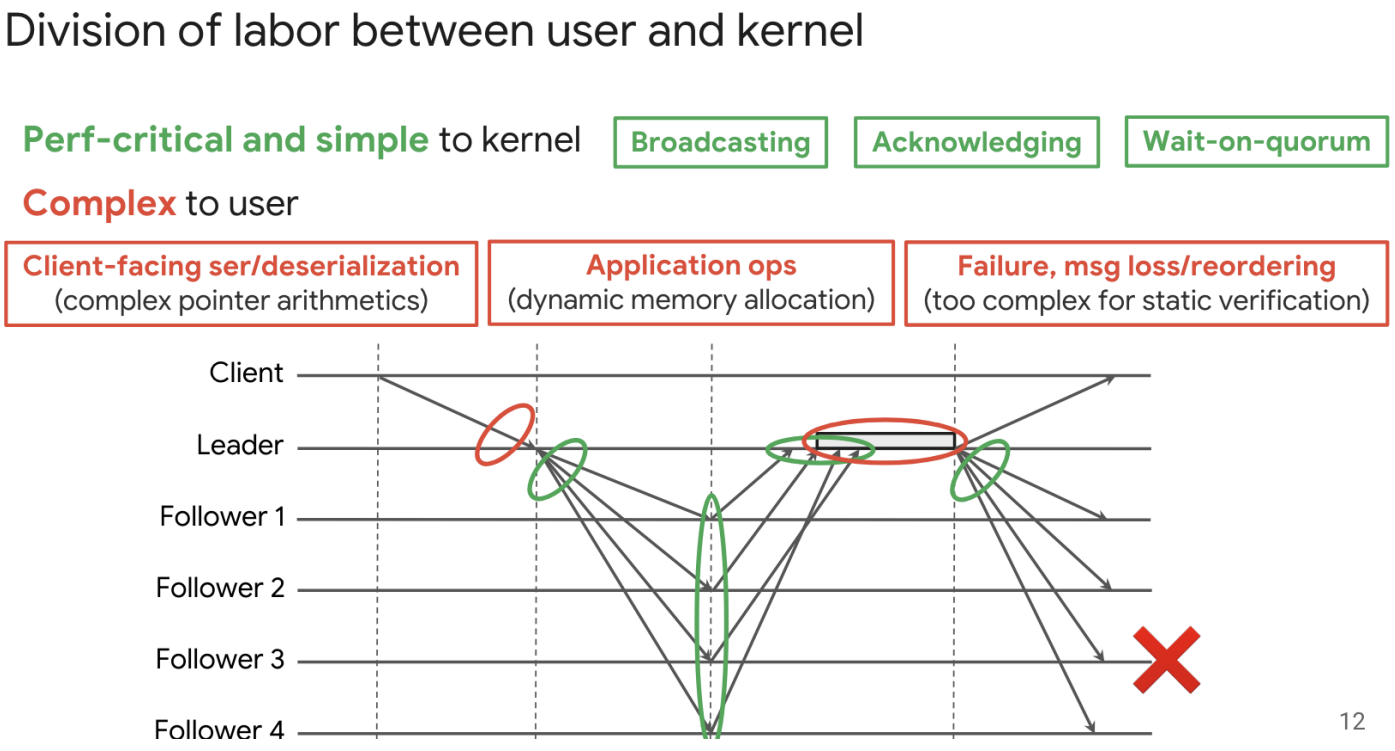
Analiza czasu CPU lidera w protokole Multi-Paxos z pięcioma replikami ujawnia, że znaczna część czasu procesora jest zużywana na przekraczanie granicy użytkownik-jądro oraz na przetwarzanie w stosie sieciowym jądra. Te obciążenia potwierdzają, że implementacja protokołów rozproszonych, używając tradycyjnego stosu sieciowego jądra, pociąga za sobą istotne koszty, które mogą być potencjalnie zmniejszone przez zastosowanie eBPF.



Model programowania eBPF jest ograniczony ze względu na statyczną weryfikację zapewniającą bezpieczeństwo:

- Ograniczona liczba instrukcji, ograniczone pętle, statyczna alokacja pamięci
- Trudności w obsłudze skomplikowanych arytmetyk wskaźników dla dostępu do pamięci

Dlatego ważne jest, aby odpowiednio podzielić pracę pomiędzy kernel i user spacem, tak aby zredukować narzut zmiany trybu, ale jednocześnie będąc w stanie zaimplementować daną funkcjonalność w eBPF. Ze względu na powyższe ograniczenia eBPF autorzy zoptymalizowali tylko niektóre części protokołu: message broadcasting, acknowledging i wait-on-quorums.



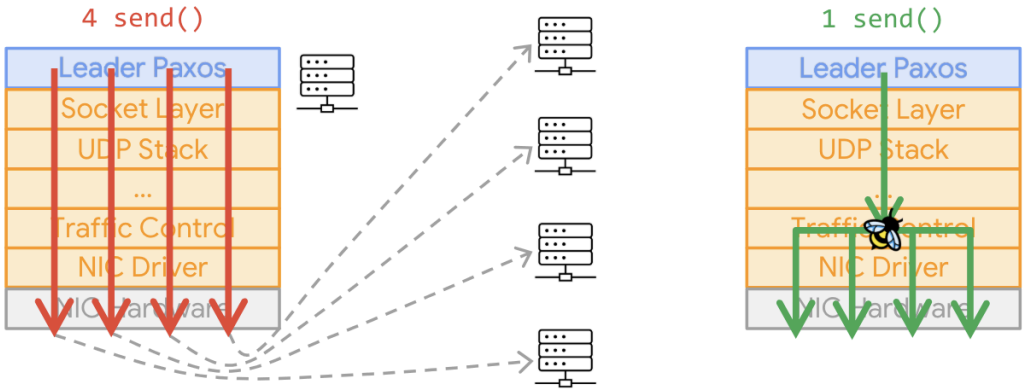
W protokołach Paxos rozgłaszanie wiadomości od jednego do wszystkich jest powszechnie stosowane. Lider wysyła wiadomości przygotowawcze do wszystkich węzłów podrzędnych, a po otrzymaniu wystarczającej liczby potwierdzeń, wysyła wiadomości zatwierdzające do wszystkich węzłów. Tradycyjna metoda implementacji tego rozgłaszania polega na wielokrotnym wysyłaniu tej samej wiadomości z przestrzeni użytkownika do różnych miejsc docelowych. Jednakże obciążenie związane z przejściami użytkownik-jądro i przekraczaniem stosu sieciowego jądra rośnie liniowo wraz z liczbą węzłów podrzędnych, co czyni węzeł lidera wąskim gardłem systemu. Electrode oferuje rozwiązanie wykorzystujące hooka eBPF w warstwie sieciowej Traffic Control. Aplikacje użytkownika mogą wywołać funkcję elec_broadcast() z określonym identyfikatorem, wiadomością i listą adresów IP docelowych, aby rozgłosić wiadomość. Program eBPF tworzy klony pakietu wiadomości za pomocą funkcji pomocniczej bpf_clone_redirect(), modyfikuje odpowiednio adresy docelowe sklonowanych pakietów i wysyła te pakiety. Korzyści z klonowania pakietów i rozgłaszania w jądrze w porównaniu z wielokrotnym wysyłaniem tej samej wiadomości w przestrzeni użytkownika polegają na tym, że konieczne jest tylko jednokrotne przekroczenie granicy użytkownik-jądro oraz przejście przez warstwy UDP i gniazd.

Electrode offload #1: message broadcasting

Perf-critical: # of context switching and stack traversing is linear to # of replicas

Simple for eBPF: TC to clone and modify packets (using `bpf_clone_redirect()`)

- Incur only once context switching and upper stack traversing
- Handle message loss in user space by resending messages (unlikely events)



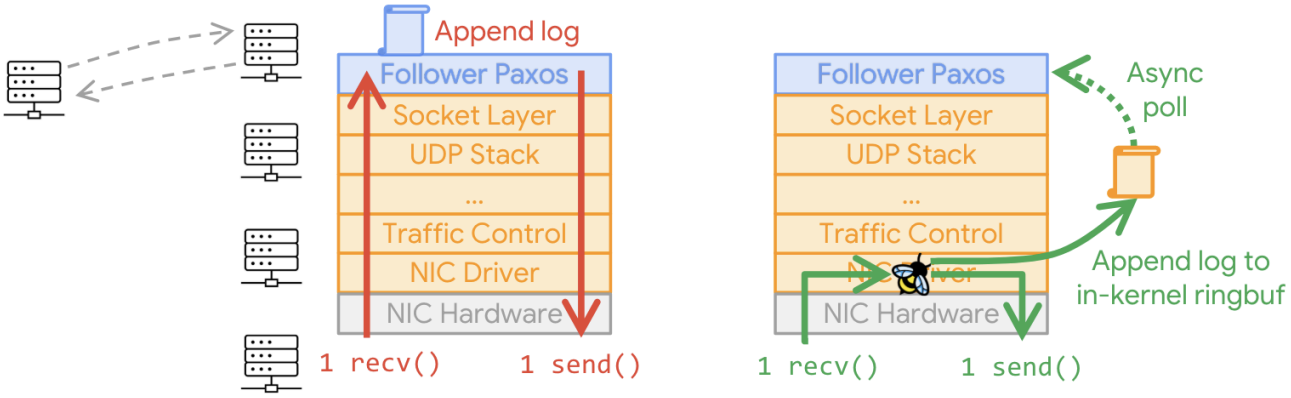
Electrode optymalizuje obsługę wiadomości przygotowawczych w węzłach podrzędnych, buforując je bezpośrednio w dzienniku w jądrze i szybko potwierdzając je węzłowi lidera. Aplikacje w przestrzeni użytkownika asynchronicznie odpytują i konsumują zbuforowane wiadomości z dziennika, korzystając z funkcji elec_poll_message(). Ta funkcja wywołuje odpowiednie wywołanie systemowe eBPF do grupowego odpytywania wiadomości, redukując obciążenie związane z przejściami między jądrem a przestrzenią użytkownika.

Electrode offload #2: fast acknowledging

Perf-critical: incurring twice the kernel latency on the critical path

Simple for eBPF: XDP to buffer log entries and quickly ack back

- Remove the kernel latency from the critical path
- Detect special cases (e.g., message loss, full buffer) and forward to user space



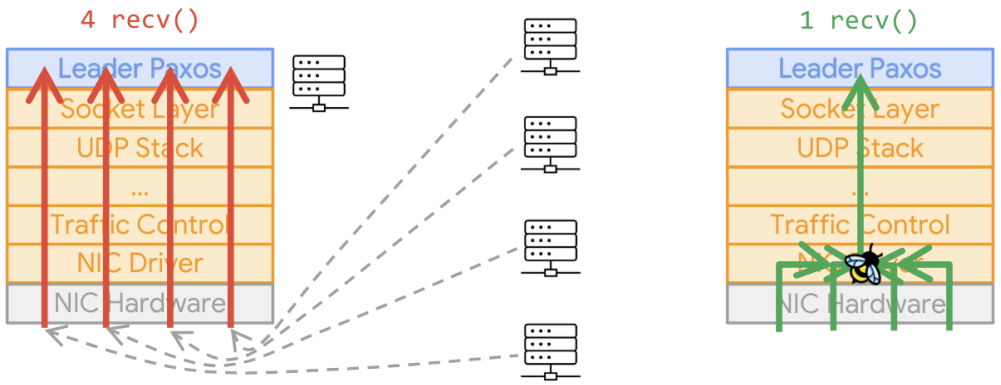
Electrode przenosi operacje oczekiwania na kworum po stronie lidera do eBPF, co wymaga tylko jednego przejścia między użytkownikiem a jądrem oraz jednego przekroczenia stosu sieciowego. Electrode utrzymuje tablicę bitsetów w eBPF, z której każdy bitset wskazuje, czy żądanie Paxos osiągnęło kworum.

Electrode offload #3: waiting on quorum

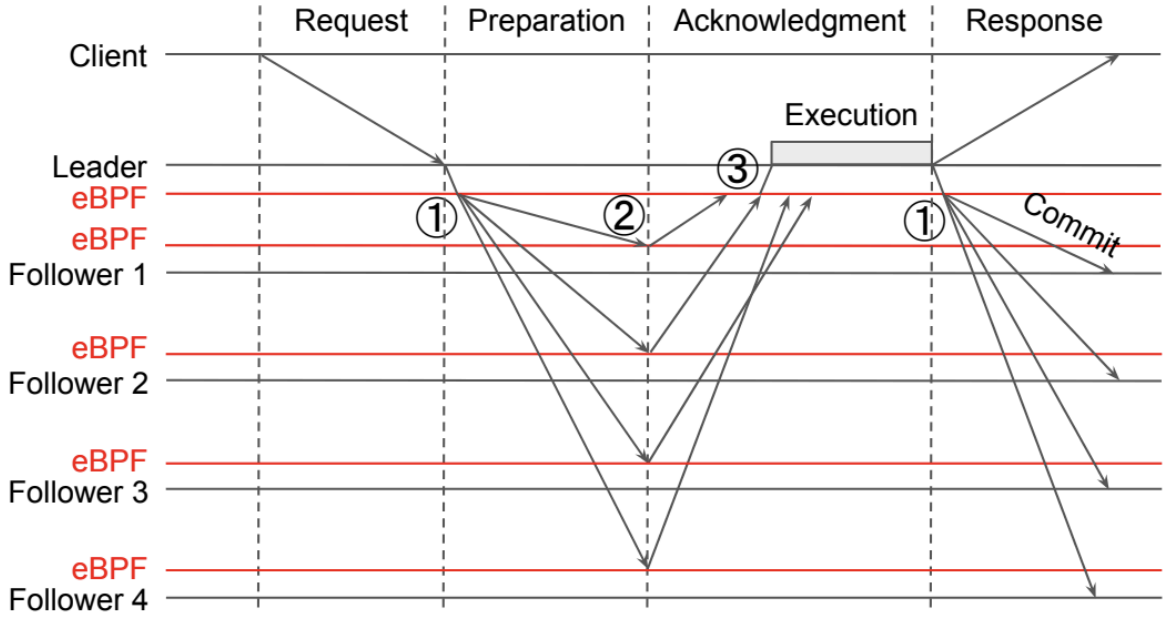
Perf-critical: leader recv ACKs from all followers, each incurring kernel overhead

Simple for eBPF: XDP to maintain # of ACKs in the driver layer

- Filter unnecessary ACKs: only the quorum-reaching ACK incurs kernel overhead
- Use bitset instead of counter to avoid double counting

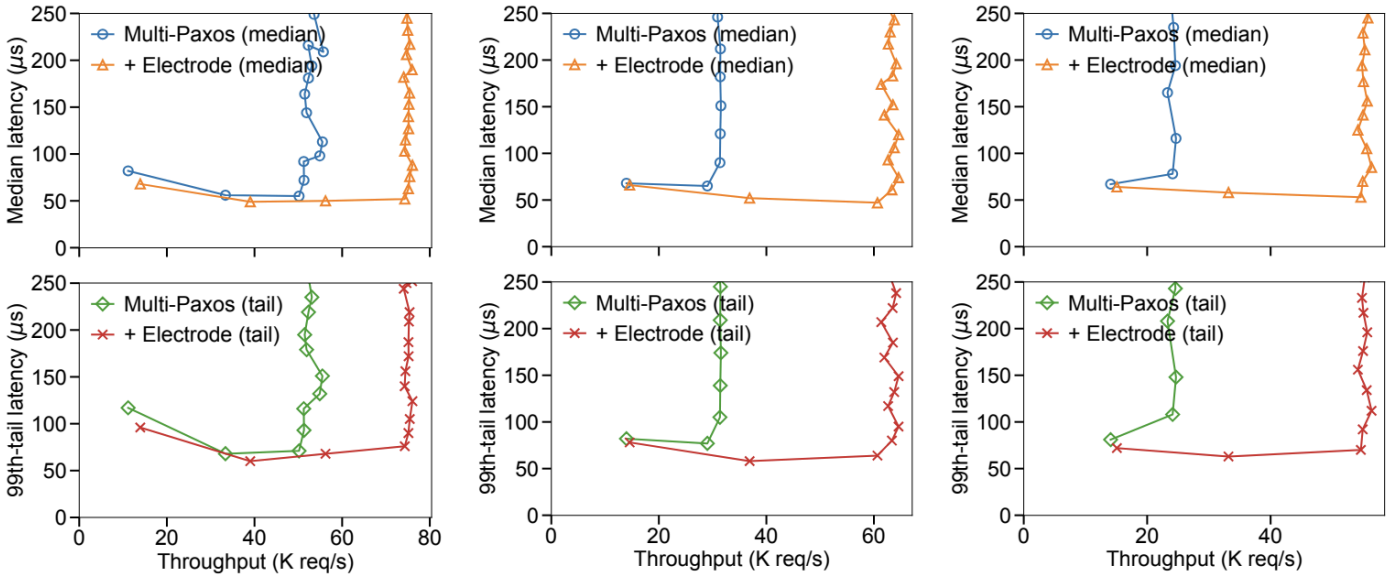


Zmniejszenie liczby zmian trybu z 14 do 5 w rozważanym zastosowaniu eBPF w protokole Paxos jest znaczącym ulepszeniem. Oznacza to, że tylko pięć razy dane przekraczają granicę między użytkownikiem a jądrem. Każde takie zmniejszenie liczby przejść między przestrzenią użytkownika a jądra jest korzystne, ponieważ przejścia te są kosztowne pod względem wydajności CPU i wpływają na ogólną latencję systemu.



b) Electrode-accelerated Multi-Paxos/Viewstamped Replication.

Zastosowanie technologii eBPF w protokole Multi-Paxos przynosi znaczące korzyści zarówno w zakresie przepustowości, jak i opóźnień. Z rysunków 5a, 5b i 5c wynika, że im większa liczba replik, tym większe są korzyści płynące z optymalizacji.



(a) 3 replicas. **(b)** 5 replicas. **(c)** 7 replicas.

Figure 5: Performance comparison of the Multi-Paxos protocol vs. Electrode-accelerated one with different numbers of replicas.

Cyprian Neugebauer
Jakub Strojewski
Michał Wójcik