Tadeusz Kobus

Replikacja Transakcyjna: Algorytmy i Własności

Streszczenie rozprawy doktorskiej

Promotor: dr hab. inż. Paweł T. Wojciechowski

Poznań · 2016

Streszczenie

Wraz z rosnącą popularnością przetwarzania w chmurze (ang. *cloud computing*), gdzie usługi (ang. *services*) działające w chmurze muszą obsługiwać ogromną liczbę użytkowników w tym samym czasie, nastąpił gwałtowny wzrost zainteresowania różnymi podejściami do *rozproszonej replikacji* (ang. *distributed replication*). Rozproszona replikacja poprawia dostępność (ang. *availability*) i niezawodność (ang. *reliability*) usługi poprzez przechowywanie danych bliżej klientów i przetwarzaniu wielu żądań klientów równolegle. W tym podejściu usługa uruchomiona jest na wielu połączonych ze sobą serwerach zwanych *replikami* (ang. *replicas*). Działania replik są koordynowane, tak by każda z replik utrzymywała spójny obraz stanu systemu rozproszonego, pomimo awarii łączy komunikacyjnych lub poszczególnych replik. Każda replika ma dostęp do lokalnej pamięci (ang. *local memory*), zaś synchronizacja replik odbywa się poprzez *pamięć współdzieloną* lub *współdzielony magazyn danych* (ang. *distributed memory*, *distributed storage*), tj. wysokopoziomową abstrakcję oferującą spójny dostęp do replikowanych danych.

Kontekst

W tej dysertacji rozważamy szczególny rodzaj replikacji zwany *replikacją transakcyjną* (ang. *transactional replication*), w której każde żądanie zgłoszone przez klienta wykonywane jest jako *atomowa transakcja* (ang. *atomic transaction*). Oznacza to, ze sekwencja operacji zdefiniowanych w żądaniu (transakcji) będzie wykonana przez zreplikowany system w semantyce *wszystko-albo-nic* (ang. *all-or-nothing semantics*). Ponadto, system odpowiedzialny jest za wykrywanie i rozwiązywanie konfliktów pomiędzy współbieżnie wykonującymi się transakcjami, które wykonują operacje na tych samych współdzielonych danych. Kod transakcji może korzystać z dodatkowych konstrukcji składniowych takich jak *rollback* czy *retry*, które pozwalają programiście na lepszą kontrolę przepływu sterowania w transakcji (*rollback* wycofuje wszystkie zmiany dokonane przez

transakcję, *retry* wycofuje transakcję i ponawia jej wykonanie od razu lub gdy spełnione są określone warunki dodatkowo zdefiniowane przez programistę).

Replikację transakcyjną usługi najłatwiej osiągnąć poprzez zbudowanie tejże usługi w oparciu o *(rozproszoną) transakcyjną platformę programistyczną* (ang. *distributed transactional programming framework*). Dzięki takiemu podejściu, programista nie musi ręcznie synchronizować współbieżnych dostępów do danych współdzielonych w oparciu o np. mechanizm *zamków* (ang. *locks*) czy *monitorów* (ang. *monitors*), których użycie stwarza problemy nawet doświadczonym programistom. Transakcyjna platforma programistyczna ukrywa całą tę złożoność przed programistą i pozwala mu wnioskować o przepływie sterowania w usłudze tak, jak gdyby wszystkie transakcje wykonywane byłyby sekwencyjnie na jednym, niezawodnym serwerze. Dzięki zapewnieniu wsparcia dla przetwarzania transakcyjnego można więc znacznie ułatwić projektowanie i rozwijanie wysoko dostępnych usług.

Iluzja jaką daje transakcyjna platforma programistyczna, polegająca na tym, że z punktu widzenia programisty transakcje wykonują się tak, jakby (logicznie) wykonywały się sekwencyjnie, wiąże się z silnymi gwarancjami, jakie ta platforma musi dawać. Gwarancje te, typowe dla tradycyjnych baz danych SQL, oryginalnie zostały sformalizowane w postaci własności poprawności (ang. *correctness property*) zwanej *(strict) serializability* (pl. *(ścisła) uszeregowalność*). Często też podejście do budowy systemów oferujących wspomniane gwarancje określane jest mianem *podejścia silnie spójnego* (ang. *the strongly consistent approach*) [1].

Uszeregowalność zazwyczaj osiągana była poprzez wykonanie wszystkich *modyfikujących transakcji* (ang. *updating transactions*) przez wyróżnioną replikę zwaną *mistrzem* (ang. *master*) i propagowanie modyfikacji wytworzonych wskutek wykonania transakcji do reszty replik zwanych *służącymi* (ang. *slaves*). Zazwyczaj każda z replik przechowywała pełną kopię bazy danych. *Transakcje niemodyfikujące* (ang. *read-only transactions*), które nie zmieniają stanu usługi, były wykonywane współbieżnie na replikach-sługach. Awaria repliki-mistrza wymagała zawieszenia przetwarzania do momentu wyłonienia nowej repliki-mistrza spośród działających replik-służących. To podejście do replikacji zostało usprawnione poprzez umożliwienie wykonania modyfikujących transakcji współbieżnie na replice-mistrzu, a później także i na różnych replikach równolegle. Wykonanie każdej transakcji koordynowane było przez jedną z replik, dzięki czemu możliwe było zapewnienie poprawności wykonania transakcji i jej późniejszego zatwierdzenia nawet w przypadku wystąpienia awarii serwerów czy też łączy komunikacyjnych. Poprzez opóźnianie wykonania niektórych operacji, bądź też wycofanie i ponowne wykonanie niektórych transakcji, protokół spójności (implementowany przez replikowany system) nie dopuszczał do wystąpienia niespójności przy współbieżnym dostępnie do współdzielonych danych. To podejście zwane jest zazwyczaj *replikacją z opóźnioną aktualizacją* (ang. *Deferred Update Replication, DUR*) [2].

Alternatywnie, baza danych mogła być replikowana przy użyciu podejścia zwanego *replikacją maszyny stanowej* (ang. *State Machine Replication, SMR*) [3] [4]

[5]. W tym podejściu identyczne kopie bazy danych uruchomione były na kilku maszynach i każda replika wykonywała wszystkie żądania przesyłane przez klientów. W ten sposób każda z replik modyfikowała swój stan w ten sam sposób, czyniąc awarie serwerów niewidocznymi dla klientów. Oczywiście wszystkie żądania musiały być deterministyczne, a także musiały być dostarczane w tej samej kolejności do wszystkich replik. W przeciwnym wypadku stan replik rozbiegłby się z czasem. W porównaniu do DUR, SMR jest dużo prostszym podejściem. Poważną wadą SMR jest jednak to, że nie pozwala na współbieżne wykonanie żądań, a więc nie może się skalować (wydajność systemu nie poprawia się wraz ze zwiększaniem liczby replik, procesorów lub rdzeni procesora). Jakkolwiek SMR nie oferuje wsparcia dla przetwarzania transakcyjnego, wykonanie pojedynczego żądania w SMR można traktować jak wykonanie prostej transakcji, która ma gwarancję zatwierdzenia.

Na początku tego milenium, pojawiły się nowe systemy baz danych, zwane bazami lub magazynami danych *NoSQL*, które powoli zaczęły zastępować tradycyjne bazy danych SQL w niektórych zastosowaniach. Magazyny NoSQL cechuje znacznie wyższa wydajność w porównaniu z tradycyjnymi bazami danych SQL, ale odbywa się to kosztem osłabionych gwarancji spójności, co utrudnia programowanie. Dużo wyższa wydajność i skalowalność tego typu systemów pozwoliła globalnie dostępnym usługom działającym w Internecie na radzenie sobie ze stale rosnącym ruchem (patrz np. [6] [7] [8] [9]).

Magazyny NoSQL zazwyczaj nie oferują wsparcia dla przetwarzania transakcyjnego i są tylko *słabo (ostatecznie) spójne* (ang. *weakly (eventually) consistent*). Oznacza to, ze klienci od czasu do czasu mogą zauważyć, że usługa działa niezgodnie z przewidzianą przez usługodawcę logiką. Zakres obserwowalnych anomalii jest szeroki: od zupełnie niegroźnych (np. kolejność wpisów na stronie portalu społecznościowego jest nieco inna niż chwilę wcześniej) do tych całkiem poważnych (klient odbiera potwierdzenie rezerwacji biletu lotniczego, po czym po pewnym czasie okazuje się, że jednak miejsce w samolocie jest już zajęte i konieczna jest rezerwacja innego lotu). Obserwowane niepożądane zachowanie usługi (niezgodne z domyślną logiką usługi) może być dalej amplifikowane przez awarie serwerów i łączy komunikacyjnych. W efekcie zapewnienie poprawnej realizacji żądań klientów wymaga od programistów więcej uwagi i implementacji dodatkowej obsługi wszystkich szczególnych przypadków. Brak jasno określonej semantyki jaką oferują magazyny NoSQL i wysoki stopień niedeterminizmu, który jest charakterystyczny dla środowiska rozproszonego, przekłada się na wysokie koszty tworzenia i utrzymywania usług zbudowanych w oparciu o takie systemy.

Jasnym jest zatem, że brak silnych gwarancji spójności oraz wsparcia dla przetwarzania transakcyjnego jest problematyczny nie tylko dla klientów, ale także i programistów. Kierujący firmami, do których należą duże usługi działające w Internecie, cały czas poszukują nowych sposobów optymalizacji kosztów działania usługi oraz poprawy jakości obsługi użytkowników (ang. *user experience*). Stąd też w ostatnich latach można obserwować wzrost zainteresowania silnie spójnymi rozwiązaniami (patrz np. [10] [11]). W tej dysertacji skupiamy

się właśnie na tego typu systemach.

W naturalny sposób silnie spójne schematy replikacji danych i usług, które badamy, czerpią z rozwiązań znanych z systemów baz danych SQL, bowiem one także są systemami rozproszonymi oferującymi semantykę transakcyjną. Drugim źródłem inspiracji są badania nad *pamięcią transakcyjną* (ang. *Transactional Memory, TM*) [12], tj. podejściem, które wykorzystuje ideę transakcji znaną z systemów bazodanowych jako mechanizm kontroli współbieżności w środowisku lokalnym (na poziomie języka programowania). Niestety żadne z istniejących rozwiązań nie może być bezpośrednio wykorzystane do rozważanych przez nas celów. Jedną z głównych przyczyn, dla których jest to niemożliwe, jest charakterystyka obciążeń (ang. *workloads*) typowa dla współczesnych replikowanych usług. Na przykład, średni czas wykonania transakcji w rozważanych przez nas systemach jest często rząd wielkości krótszy niż w w systemach bazodanowych i rząd wielkości dłuższy niż w systemach pamięci transakcyjnej [13]. Także logika współczesnych replikowanych usług (lub fragment tejże logiki) często nie może być łatwo wyrażona przy pomocy SQL, tj. języka tradycyjnych baz danych. Często lepszym podejściem jest wykorzystanie interfejsu przypominającego interfejs systemów pamięci transakcyjnych, gdzie transakcja traktowana jest jako wysokopoziomowa konstrukcja języka programowania, w ramach której można definiować dowolny kod, a w szczególności taki, który wywołuje złożone metody czy operacje na obiektach współdzielonych (patrz np. [14] [15]). W naszej pracy skupiamy się na takim właśnie podejściu, gdyż jest ono bardziej ogólne.

Z uwagi na różnice w oferowanych interfejsach, własności poprawności używane w kontekście systemów baz danych (takie jak *recoverability*, *avoiding cascading aborts*, *strictness* [16] czy warianty wspomnianej już własności uszeregowalności) nie mogą być stosowane do formalizacji gwarancji oferowanych przez współczesne silnie spójne schematy replikacji. Niestety nie mają zastosowania w naszej pracy także własności znane ze świata systemów pamięci transakcyjnych (takie jak różne warianty *opacity* [17] [18] [19], *TMS1* [20] czy *TMS2* [20]). Jest tak, ponieważ własności te zostały zaprojektowane w kontekście lokalnego środowiska, gdzie pewne gwarancje dotyczące uszeregowania transakcji (takie jak uwzględnienie ograniczeń *czasu rzeczywistego*, ang. *real-time*) są naturalne i relatywnie niedrogie do zapewnienia. Natomiast w środowisku rozproszonym wspomniane gwarancje muszą często być osłabione w odniesieniu do niektórych typów żądań (transakcji), np. żądań niemodyfikujących (ang. *read-only*). Dlatego też formalizacja semantyki replikowanych systemów, które rozważane są w tej dysertacji, wymaga zdefiniowania nowych własności poprawności.

Można zatem stwierdzić, że pomimo na powrót rosnącej popularności silnie spójnych schematów replikacji, podstawy teoretyczne tego typu rozwiązań nie są jeszcze dobrze poznane i potrzeba więcej badań w tej dziedzinie. Na przykład brak w istniejącej literaturze szczegółowego porównania podstawowych schematów replikacji takich jak SMR i DUR, zarówno pod kątem semantyki jak i wydajności. Jest to zaskakujące, ponieważ zarówno SMR jak i DUR stanowią podstawę wielu innych, bardziej skomplikowanych schematów replikacji (patrz

np. [21] [22] [23] [24] [25] [26]). Dopiero gdy odkryjemy różnice między tymi podejściami i jasno określimy ich silne i słabe strony, będziemy mogli zaproponować nowe schematy replikacji, które będą odpowiadać aktualnym potrzebom oraz w pełni wykorzystywać możliwości współczesnych, wysoce równoległych architektur systemów komputerowych.

Cele i kontrybucje

Biorąc pod uwagę powyższe motywacje, w następujący sposób formułujemy główną tezę dysertacji:

> *Jest możliwe stworzenie schematu replikacji usług i danych, który oferuje bogatą semantykę transakcyjną, daje silne gwarancje spójności oraz cechuje go wysoka wydajność dla różnych typów obciążeń.*

Poniżej krótko podsumowujemy kontrybucje ujęte w dysertacji:

1. **Nowe własności poprawności dla silnie spójnych replikowanych systemów**. Definiujemy $\diamond$-*opacity* i $\diamond$-*linearizability*, dwie rodziny własności poprawności zaprojektowane dla silnie spójnych replikowanych systemów. Nasze własności wywodzą się z *opacity* (pl. *nieprzezroczystości*) i *linearizability* (pl. *liniowości*)–dwóch dobrze znanych własności poprawności zdefiniowanych dla systemów transakcyjnych i systemów modelowanych jako obiekty współdzielone [17] [27]. Dzięki zaproponowanym nowym własnościom, możemy sformalizować gwarancje oferowane przez różne schematy replikacji, które oferują (lub nie) semantykę transakcyjną oraz implementują różnego rodzaju optymalizacje, takie jak np. wykonanie żądań niemodyfikujących przez pojedynczą replikę, bez dodatkowej synchronizacji między replikami. Dowodzimy, że wszystkie własności z rodzin $\diamond$-opacity i $\diamond$-linearizability są własnościami bezpieczeństwa (tzn. są niepuste, prefiksowo-domknięte i domknięte, ang. *non-empty*, *prefix-closed*, *limit-closed*). Określamy również formalny związek pomiędzy zaproponowanymi rodzinami własności. Pokazujemy, że gdy żądania są wykonane w systemie gwarantującym $\diamond$-opacity i transakcje są niewidoczne dla klientów (tzn. klienci nie widzą pośrednich wyników wykonania transakcji i powiadamiani są o wyniku przetwarzania tylko przy zatwierdzaniu lub wycofywaniu transakcji), wtedy system gwarantuje $\diamond$-linearizability. W szczególności pokazujemy związek pomiędzy opacity i linearizability w ich oryginalnych definicjach (wg. naszej najlepszej wiedzy, jest to pierwszy rezultat tego typu).
2. **Szczegółowe porównanie SMR i DUR**. Porównujemy SMR i DUR zarówno pod względem oferowanej semantyki jak i wydajności. Formalnie dowodzimy poprawność obu podejść i pokazujemy, że SMR gwarantuje *real-time linearizability* (własność z rodziny $\diamond$-linearizability), podczas gdy

DUR gwarantuje *update-real-time opacity* (własność z rodziny $\diamond$-opacity). Dowodzimy także, że DUR gwarantuje *update-real-time linearizability*, gdy transakcje są ukryte przed klientami. Tym samym pokazujemy, że gwarancje oferowane przez DUR są ściśle słabsze niż gwarancje oferowane przez SMR. Rozważamy również *SMR with Locks (LSMR)*, tj. SMR implementujące optymalizację polegającą na tym, że niemodyfikujące żądania są wykonywane tylko przez pojedynczą replikę. Określamy dokładnie jakie skutki dla oferowanych gwarancji ma wprowadzenie tej optymalizacji i formalnie pokazujemy, że LSMR oferuje gwarancje ściśle słabsze niż LSMR, ale ściśle silniejsze niż DUR. Wyniki przeprowadzonej przez nas ewaluacji eksperymentalnej, pokazują silne i słabe strony SMR i DUR w przypadku różnego rodzaju obciążeń. Głównym wnioskiem płynącym z porównania wydajności obu podejść jest to, że żadne podejście nie jest ściśle lepsze od drugiego w ogólnym przypadku. Ten rezultat może zaskakiwać, ponieważ jedynie w przypadku DUR wydajność może potencjalnie rosnąć wraz ze zwiększającą się liczbą replik biorących udział w przetwarzaniu (SMR wykonuje wszystkie żądania sekwencyjnie, natomiast LSMR, tj. zoptymalizowany wariant SMR, pozwala na równoległe przetwarzanie żądań tylko w przypadku żądań niemodyfikujących).

3. **Nowy, silnie spójny schemat replikacji transakcyjnej**. Proponujemy nowy schemat replikacji zwany *hybrydową replikacją transakcyjną* (ang. *Hybrid Transactional Replication, HTR*). HTR łączy SMR i DUR dla lepszej wydajności, skalowalności i bogatszej semantyki. Formalnie dowodzimy, że HTR oferuje gwarancje podobne do DUR. Jak pokazujemy w testach ewaluacyjnych, HTR działa dobrze przy różnego rodzaju obciążeniach. W szczególności, HTR dobrze radzi sobie z obciążeniami, o których wiadomo, że są problematyczne dla SMR czy DUR (np. obciążenia charakteryzujące się długimi czasami wykonań żądań w przypadku SMR i obciążenia, w których występuje wysokie współzawodnictwo w dostępie do tych samych danych w przypadku DUR). W niektórych przypadkach, HTR pozwala na osiągnięcie nawet 50% lepszej wydajności niż w przypadku uruchomienia HTR symulującego działanie DUR lub LSMR (wszystkie żądania modyfikujące są wykonywane w sposób, który przypomina wykonanie transakcji w DUR lub żądań w LSMR). HTR pozwala na definiowanie polityk, dzięki którym możliwe jest dostosowanie działania systemu do spodziewanego obciążenia i tym samym adaptacja do zmieniających się warunków brzegowych. Przedstawiamy szereg technik przydatnych przy tworzeniu polityk, a także proponujemy politykę wykorzystującą mechanizmy uczenia maszynowego, która ułatwia pracę programiście i pozwala na automatyczne dostosowywanie działania systemu do zmieniających się warunków.

Powyższym kontrybucjom, które też opisujemy nieco bardziej szczegółowo w dalszej części streszczenia, poświęcone są Rozdziały 4, 5 i 6 dysertacji. W pozostałych rozdziałach dysertacji przedstawiamy kontekst i tezę dysertacji (Roz-

dział 1), przegląd literatury związanej z tematyką pracy (Rozdział 2), definiujemy model rozważanych systemów (Rozdział 3). Podsumowujemy dysertację w Rozdziale 7.

Nowe własności poprawności dla systemów zreplikowanych

Gwarancja wykonania żądań klientów z uwzględnieniem ograniczeń *czasu rzeczywistego* (ang. *real-time*) jest często pożądaną cechą systemów rozproszonych. Gwarancja czasu rzeczywistego oznacza, że gdy wykonanie jednego żądania kończy się zanim rozpocznie się wykonanie innego żądania (porównując np. czas zegarowy obu zdarzeń), efekty wykonania pierwszego żądania są zawsze widoczne dla wykonania drugiego żądania. Zapewnienie takiej (intuicyjnej) gwarancji w środowisku rozproszonym nie jest proste i często okazuje się bardzo kosztowne. Jest tak dlatego, ponieważ uwzględnienie ograniczeń czasu rzeczywistego wymaga synchronizacji między procesami (serwerami) w przypadku wykonania każdego żądania. Dlatego też replikowane usługi często osłabiają ograniczenie czasu rzeczywistego w przypadku niektórych typów żądań, na przykład żądań *niemodyfikujących* (żądań, które nie wykonały żadnych operacji modyfikujących jak np. operacja zapisu, lub żądań *wycofanych*, ang. *aborted*, *rolled back*). Wykonanie tego typu żądań nie wpływa na stan usługi, dlatego też ich wykonanie nie musi uwzględniać ograniczeń czasu rzeczywistego względem reszty żądań, które zmieniają stan systemu. W ten sposób wydajność systemu może znacznie wzrosnąć, w szczególności, gdy żądania niemodyfikujące stanowią większość żądań przetwarzanych przez system.

Brak gwarancji uwzględnienia czasu rzeczywistego (dla chociażby niektórych typów żądań) z pozoru wydaje się dość błahy. Jednak, jak pokazujemy w Sekcji 4.1 na przykładzie DUR, brak owych gwarancji ma istotne konsekwencje. Muszą one być być brane pod uwagę przez programistę replikowanej usługi, gdy klienci usługi mogą komunikować się między sobą nie tylko poprzez zreplikowany system, ale także innymi kanałami (np. w szczególności poprzez zewnętrzne usługi).

Jak się okazuje, semantyka takich schematów replikacji jak DUR, nie jest właściwie opisana przez żadną z istniejących własności poprawności (patrz Sekcja 2.2.1). Niektóre własności, które nie uwzględniają ograniczeń czasu rzeczywistego są za słabe. Na przykład wspomniane już wcześniej *serializability* (pl. *uszeregowalność*) [1] definiuje ograniczenia tylko dla zatwierdzonych transakcji i nie określa gwarancji dla transakcji żywych lub wycofanych. *Update serializability* [28] czy *extended update serializability* [29] dopuszczają *obserwowanie* przez różne procesy różnej historii zatwierdzeń modyfikujących transakcji w systemie. Inne znane własności takie jak *opacity* (pl. *nieprzezroczystość*) [17] czy *TMS1* [20] są zbyt silne, ponieważ wymagają by ograniczenie czasu rzeczywistego było zawsze przestrzegane (dla wszystkich typów transakcji). Warto zauważyć, że własności takie jak opacity czy TMS1 były zaproponowane dla systemów *pa-*

mięci transakcyjnych (ang. *transactional memory, TM systems*), tj. mechanizmów synchronizacji współbieżnego dostępu do danych mającego stanowić alternatywę dla zamków, patrz np. [12]. Dlatego też ograniczenie czasu rzeczywistego jest w tym wypadku naturalne. Jakkolwiek z uwagi na to, że systemy pamięci transakcyjnej funkcjonują w środowisku lokalnym a nie rozproszonym, ograniczenie czasu rzeczywistego jest relatywnie proste do zagwarantowania.

$\diamond$-*opacity*, które definiujemy w Sekcji 4.3, jest rodziną blisko powiązanych ze sobą własności poprawności opartych na opacity. Własności te osłabiają w systematyczny sposób ograniczenie czasu rzeczywistego dotyczące uszeregowania transakcji w opacity. Ogólnie mówiąc, system który gwarantuje którąkolwiek własność z rodziny $\diamond$-opacity, zachowuje się tak, jak gdyby wszystkie transakcje (a więc również żywe i wycofane transakcje) były wykonywane sekwencyjnie. Wymagania dotyczące uszeregowania transakcji, które to uszeregowanie obserwuje klient, zależą od rozważanej własności. W skrajnych przypadkach ograniczenie czasu rzeczywistego musi być zawsze przestrzegane (zgodnie z *real-time opacity*) albo nigdy nie musi być brane pod uwagę (zgodnie z *arbitrary order opacity*). Oznacza to, że najsilniejsza własność z rodziny $\diamond$-opacity jest tożsama z oryginalną definicją opacity w jej prefiksowo-zamkniętej definicji [17]. Z drugiej strony, arbitrary order opacity przypomina serializability, ale jest zdefiniowane nie tylko dla zatwierdzonych transakcji, ale także dla transakcji żywych i wycofanych. Obecnie $\diamond$-opacity definiuje jeszcze cztery inne własności, słabsze od real-time opacity ale silniejsze niż arbitrary order opacity: *commit-real-time opacity*, *write-real-time opacity*, *update-real-time opacity* i *program order opacity*. Formalnie dowodzimy, że DUR spełnia update-real-time opacity. Własność ta pozwala transakcjom niemodyfikującym oraz transakcjom wycofanym na działanie na stanie systemu, który nie jest najświeższy, ale jest nadal spójny (patrz Sekcja 5.3.3). Write-real-time opacity i commit-real-time opacity są pośrednimi własnościami, dzięki którym możemy porównywać gwarancje oferowane przez transakcyjne i nietransakcyjne schematy replikacji (patrz niżej). Program order opacity jest własnością podobną do virtual time opacity [30], ale zdefiniowaną w oparciu o zbiór pojęć podstawowych wykorzystanych w oryginalnej definicji opacity (patrz także Sekcja 2.2.1).

Wraz z $\diamond$-opacity, definiujemy rodzinę własności zwaną $\diamond$-*linearizability* i bazującą na *linearizability* (pl. *liniowość*) [27], czyli dobrze znanej własności zwykle wykorzystywanej do formalizacji semantyki współbieżnych struktur danych. Własności z rodziny $\diamond$-linearizability pozwalają określić gwarancje oferowane przez silnie spójne systemy, w których przetwarzanie transakcyjne następuje w sposób *niewidoczny* dla klientów (klienci nigdy nie widzą pośrednich wyników wykonania transakcji). Własności te mogą być również używane do zdefiniowania gwarancji systemów, które nie oferują semantyki transakcyjnej. W szczególności, nowe własności pozwalają opisać gwarancje różnych wariantów SMR. W zależności od implementowanej optymalizacji, SMR spełnia real-time linearizability lub też słabsze własności takie jak *write-real-time linearizability* (patrz Sekcje 5.1.3 i 5.2.3). Ponadto, jak dowodzimy, $\diamond$-linearizability zachowuje dwie istotne własności linearizability: *lokalność* (ang. *locality*) i *nieblokowanie* (ang. *non-*

blocking).

W Sekcji 4.5 przedstawiamy formalny rezultat dotyczący relacji pomiędzy $\diamond$-opacity i $\diamond$-linearizability. Ogólnie mówiąc, pokazujemy, że jeśli transakcje są niewidoczne dla klientów, replikowany system spełniający $\diamond$-opacity, spełnia również $\diamond$-linearizability. Rezultat ten ustanawia formalny związek między opacity i linearizability (w ich oryginalnych definicjach) i pozwala bezpośrednio porównać gwarancje systemów takich jak DUR i SMR. Ponadto *obiekt-brama* (ang. *gateway object*), który zdefiniowaliśmy w celu ustanowienia związku między dwiema rodzinami zaproponowanych własności, jest na tyle ogólny, że może być używany do porównania innych (transakcyjnych lub nietransakcyjnych) własności poprawności.

Porównanie podstawowych schematów replikacji

Dwoma podstawowymi silnie spójnymi schematami replikacji są wspomniane już wcześniej podejścia zwane *replikacją maszyny stanowej* (ang. *State Machine Replication, SMR*) [3] [4] [5] i *replikacją z opóźnioną aktualizacją* (ang. *Deferred Update Replication, DUR*) [2]. Oba podejścia można implementować przy użyciu różnych *rozproszonych protokołów uzgadniania* (ang. *distributed agreement protocols*). My skupiamy się na implementacjach zbudowanych w oparciu o protokół *rozgłaszania z globalnym uporządkowaniem* (ang. *Total Order Broadcast, TOB* [31]).

Fundamentalna różnica pomiędzy SMR i DUR polega na innej kolejności synchronizacji serwerów (replik), na których uruchomiona jest replikowana usługa, oraz wykonania żądań. W SMR repliki komunikują się przed wykonaniem żądania poprzez rozgłoszenie żądania przy użyciu TOB. Następnie każda z replik niezależnie wykonuje żądanie, tym samym uaktualniając swój stan w podobny sposób (o ile wykonanie żądania jest deterministyczne). Natomiast w DUR, jak już wspomnieliśmy wcześniej, żądanie jest wykonywane optymistycznie jako atomowa transakcja na pojedynczej replice. Dopiero po zakończeniu wykonania transakcji, modyfikacje utworzone w trakcie wykonania są rozgłaszane przy pomocy TOB do wszystkich replik, tak by uaktualniły one swój stan (ale tylko wtedy, gdy procedura certyfikacji transakcji zakończy się powodzeniem).

Różnice w sposobie wykonania żądań w SMR i DUR skutkują znacznymi rozbieżnościami w wydajności obu podejść wobec różnego rodzaju obciążeń. Na przykład SMR jest bardzo wrażliwe na obciążenia charakteryzujące się dużą liczbą żądań wymagających długiego czasu wykonania. Jest to zrozumiałe, ponieważ SMR wykonuje wszystkie żądania sekwencyjnie. DUR natomiast radzi sobie z tego typu obciążeniami bardzo dobrze. Dzięki temu, że w DUR każde żądanie wykonywane jest (jako transakcja) na pojedynczym serwerze, wykonanie żądań można zrównoleglić przetwarzając je na nowoczesnych wielordzeniowych procesorach i wielu replikach jednocześnie. Z drugiej strony DUR źle radzi sobie z obciążeniami, w których można obserwować wysoki stopień współ-

zawodnictwa w dostępie do tych samych danych (ang. *high contention levels*). Do takiej sytuacji dochodzi na przykład wtedy, gdy wiele transakcji w tym samym momencie chce modyfikować te same dane. W takim wypadku wiele transakcji musi być wycofanych i ponowionych, co odbija się negatywnie na wydajności i skalowalności DUR. Ponieważ w SMR wszystkie żądania wykonywane są sekwencyjnie, to czy żądania często odwołują się do tych samych danych nie wpływa zasadniczo na wydajność SMR.

Głównym wnioskiem płynącym z badań ewaluacyjnych SMR i DUR, których wyniki przedstawiamy i omawiamy w Sekcji 5.4, jest to, że żadne z podejść nie jest ściśle lepsze od drugiego. Jest to wniosek zaskakujący, ponieważ w przeciwieństwie do DUR, SMR nie może skalować się wraz z rosnącą liczbą replik (a więc także coraz większą dostępną mocą obliczeniową). Dla danego schematu replikacji, obciążenia i rozmiaru klastra, niemal zawsze można jednoznacznie wskazać czynnik decydujący o ograniczonej przepustowości systemu, tzn. relatywnie długie czasy wykonania żądań (które dominują czasy synchronizacji replik) lub długie czasy synchronizacji replik (które dominują czasy wykonania żądań). Niestety, jak pokazujemy, ów dominujący czynnik bywa różny nie tylko dla różnego rodzaju obciążeń, ale czasami zmienia się wraz ze zmianą konfiguracji klastra (np. włączeniem do przetwarzania dodatkowych replik). Tak więc wybór sposobu replikacji usługi powinien być uzależniony nie tylko od spodziewanego obciążenia, ale i rozmiaru klastra.

W kwestii semantyki oba podejścia również znacznie się różnią. DUR oferuje semantykę transakcyjną, a zatem kod żądania (wykonywanego jako transakcja) może korzystać z takich konstrukcji składniowych jak *rollback* czy *retry*, które dają programiście kontrolę nad przepływem sterowania w transakcji. Ponieważ transakcje w DUR mogą zostać w dowolnym momencie przerwane i powtórzone (ze względu na wykrywane przez system konflikty przy współbieżnym dostępie do danych), kod takiej transakcji nie może zawierać tzw. *operacji niewycofywalnych* (ang. *irrevocable operations*), a więc takich, których efektów nie będzie można cofnąć, jak np. w przypadku wywołań systemowych czy operacji wejścia/wyjścia. Kod transakcji może natomiast zawierać niedeterministyczne operacje, ponieważ wykonanie transakcji odbywa w sposób optymistyczny na pojedynczym serwerze. Nie jest tak w przypadku żądań wykonywanych przez SMR. Ponieważ każde żądanie jest wykonywane niezależnie przez każdą replikę, kod żądania musi być deterministyczny. W przeciwnym wypadku stan replik uległby rozbiegnięciu.

Gwarancje oferowane przez SMR i DUR można sformalizować przy pomocy własności własności należących do rodzin $\diamond$-linearizability i $\diamond$-opacity, tj. *real-time linearizability* i *update-real-time linearizability* (patrz Sekcje 5.1.3 i 5.3.3). Znając relację pomiędzy rodzinami własności, możemy pokazać, że DUR oferuje ściśle słabsze gwarancje niż SMR. W Sekcji 5.2.3 badamy gwarancje LSMR, a więc SMR, który implementuje optymalizację polegającą na tym, że żądania, o których z góry wiadomo, że nie zmienią stanu usługi, można wykonywać tylko na pojedynczej replice, bez dodatkowej synchronizacji z innymi replikami. Jak pokazujemy, LSMR gwarantuje *write-real-time linearizability*, a więc oferuje gwaran-

cje ściśle silniejsze niż DUR ale ściśle słabsze niż SMR. Widać zatem, że wybór sposobu replikacji usługi nie tylko powinien uwzględniać kwestie wydajności, ale także oczekiwanych gwarancji na wykonanie żądań (znacznie różniących się między SMR i DUR).

Nowy silnie spójny transakcyjny schemat replikacji

W dysertacji opisujemy zaproponowane przez nas nowe podejście do replikacji zwane *hybrydową replikacją transakcyjną* (ang. *Hybrid Transactional Replication, HTR*). W HTR każde żądanie jest wykonywane jako transakcja w jednym z dwóch trybów: *deferred update (DU)* lub *state machine (SM)*. W pierwszym przypadku wykonanie transakcji odbywa się dokładnie tak jak w DUR. Tym samym, HTR wykonujący wszystkie żądania jako transakcje w trybie DU jest tożsamy z DUR. W przypadku wykonania transakcji w trybie SM, najpierw transakcja jest rozgłaszana przy pomocy TOB do wszystkich replik i wtedy każda z replik niezależnie wykonuje transakcję. Tak więc wykonanie transakcji w trybie SM jest bardzo podobne do wykonania żądania w SMR, ale z dodatkowym wsparciem dla przetwarzania transakcyjnego (kod transakcji SM może zawierać konstrukcje składniowe takie jak *rollback* i *retry*). HTR pozwala na współbieżne wykonywanie wielu transakcji w trybie DU i jednej transakcji w trybie SM. W HTR wykonanie transakcji SM jest serializowane w jednym wątku z certyfikacją transakcji wykonanych w trybie DU i ewentualnym późniejszym uaktualnianiem stanu repliki (gdy procedura certyfikacji transakcji zakończy się sukcesem).

Semantyka obu trybów wykonania transakcji jest podobna do semantyki wykonania transakcji w DUR i żądań w SMR. Transakcje wykonywane w trybie DU nie mogą zawierać operacji niewycofywalnych, ale mogą wykonywać niedeterministyczny kod. Natomiast transakcje wykonywane w trybie SM muszą być deterministyczne, ale mogą zawierać operacje niewycofywalne, gdyż takie transakcje mają gwarancję zatwierdzenia. W efekcie, HTR oferuje semantykę bogatszą niż DUR i SMR.

Jak formalnie dowodzimy w Sekcji 6.4, HTR spełnia update-real-time opacity, a więc oferuje te same gwarancje spójności co DUR. Dodatkowo, jeśli wszystkie transakcje wykonywane w HTR są wykonywane w trybie SM, to HTR spełnia real-time opacity, własność analogiczną do tej, którą gwarantuje SMR.

Wybór trybu wykonania transakcji nie musi być wyłącznie uzależniony od tego, jakie gwarancje oczekiwane są w odniesieniu do wykonania transakcji. Na przykład transakcje, które w trybie DU wielokrotnie musiałyby być wycofywane z uwagi na wykrycie konfliktów przy współbieżnym dostępie do danych (np. operacja zmieniania rozmiaru tablicy z haszowaniem), często lepiej jest wykonać w trybie SM. Wybór tego trybu wykonania transakcji daje pewność, że zatwierdzenie transakcji powiedzie się. W trybie SM warto także wykonywać transakcje, które dokonują wielu modyfikacji. W przypadku wykonania transakcji w trybie SM zmiany są tworzone niezależnie przez wszystkie repliki i nie

muszą być transmitowane przez sieć (tak jakby to było konieczne w przypadku wykonania transakcji w trybie DU). Natomiast tryb DU sprawdza się lepiej w przypadku transakcji, które sporadycznie odwołują się do tych samych danych i których wykonanie trwa stosunkowo długo. Wtedy takie transakcje można wykonywać w pełni równolegle.

Decyzja na temat tego, w którym trybie wykonać transakcję jest podejmowana przed każdym wykonaniem transakcji przez tzw. *wyrocznię* (ang. *oracle*). Naturalnie transakcja, o której z góry wiadomo, że nie zmodyfikuje stanu systemu jest wykonywana w trybie DU przez pojedynczą replikę. W celu optymalizacji wykorzystania zasobów, wyrocznia podejmuje decyzję dotyczącą trybu wykonania potencjalnie modyfikującej transakcji w oparciu o aktualny stan systemu, rodzaj wykonywanej transakcji oraz zdefiniowaną wcześniej politykę. W szczególności wyrocznia bierze pod uwagę obciążenie serwerów, stopień nasycenia sieci, średnie czasy wykonania i zatwierdzania transakcji oraz stopień współzawodnictwa transakcji w dostępie do tych samych danych. Polityka, którą dostarcza programista, powinna odpowiadać oczekiwanemu obciążeniu i może wykorzystywać mechanizmy uczenia maszynowego (ang. *machine learning*), dzięki czemu system może automatycznie dostosowywać się do zmieniającego się obciążenia. Przykładową politykę korzystającą z mechanizmów uczenia maszynowego przedstawiamy i ewaluujemy w rozprawie (patrz Sekcje 6.6 i 6.7). Zaprezentowane rozwiązanie inspirowane jest dobrze znanymi w środowisku uczenia maszynowego algorytmami rozwiązującymi *problem wielorękiego bandyty* (ang. *multi-armed bandit problem*) [32] [33].

Wyniki testów jasno demonstrują korzyści płynące z łączenia dwóch podejść do replikacji w HTR. Nasz system radzi sobie dobrze z różnego rodzaju obciążeniami, w tym z obciążeniami, o których wiadomo, że będą problematyczne dla DUR czy SMR. Wydajność HTR jest co najmniej tak dobra jak wydajność DUR czy SMR z osobna. Przy niektórych typach obciążeń wydajność oferowana przez HTR jest nawet 50% lepsza niż w przypadku wykonania wszystkich modyfikujących transakcji w trybie DU czy SM (co odpowiada wykonaniu transakcji w DUR czy wykonaniu żądań w LSMR, a więc zoptymalizowanej wersji SMR). Przeprowadzone przez nas testy pokazują również, że wyrocznia wykorzystująca mechanizmy uczenia maszynowego niemal natychmiast reaguje na zmiany obciążenia i działa bardzo dobrze bez względu na rozmiar klastra, na którym uruchomiona jest zreplikowana usługa.

Podsumowanie

Główna teza dysertacji dotyczyła zaproponowania nowego silnie spójnego schematu replikacji o semantyce transakcyjnej, który oferowałby wysoką wydajność dla różnych typów obciążeń. W celu dowiedzenia tezy dysertacji wykonaliśmy badania, które podsumowujemy poniżej.

Po pierwsze zaproponowaliśmy $\diamond$-opacity i $\diamond$-linearizability, dwie nowe ro-

dziny własności poprawności bazujące na dobrze znanych własnościach poprawności wykorzystywanych w kontekście systemów transakcyjnych i systemów modelowanych jako obiekty współdzielone. Zaproponowane własności systematyzują klasy różnych systemów replikacji, umożliwiają formalną analizę oferowanych przez nich gwarancji, a także ich porównanie na płaszczyźnie semantyki. W szczególności, dzięki zaproponowanym w dysertacji nowym własnościom i udowodnionej formalnej relacji między nimi, możliwe jest porównanie gwarancji oferowanych przez silnie spójne systemy, które oferują bądź nie semantykę transakcyjną, a także możliwe jest porównanie bazowych własności (tj. opacity i linearizability) w ich oryginalnych definicjach.

Po drugie dokonaliśmy dogłębnego porównania SMR i DUR, dwóch podstawowych silnie spójnych schematów replikacji. W tym celu, korzystając z wcześniej zdefiniowanych własności, formalnie udowodniliśmy oferowane przez te podejścia gwarancje. Następnie nakreśliliśmy różnice w semantyce obu podejść a także porównaliśmy SMR i DUR eksperymentalnie. Co ciekawe, żadne z podejść nie okazało się ściśle lepsze od drugiego. Wybór schematu replikacji powinien zatem zależeć nie tylko od oczekiwanych gwarancji odnośnie wykonywanych żądań, ale także od spodziewanego rodzaju obciążenia.

Po trzecie zaproponowaliśmy nowe, silnie spójne podejście do replikacji zwane hybrydową replikacją transakcyjną, które łączy w jeden schemat SMR i DUR. HTR zachowuje bogatą semantykę transakcyjną DUR i rozszerza ją o wsparcie dla operacji niewycofywalnych. Jak formalnie dowodzimy, HTR oferuje te same gwarancje na wykonanie transakcji jak DUR (a w przypadku wykonania wszystkich żądań w trybie SM oferowane gwarancje są nawet silniejsze). Ponadto, HTR cechuje bardzo dobra wydajność w szerokim zakresie obciążeń, w tym obciążeń będących problematycznymi dla SMR czy DUR. Co więcej, możliwość wykonania części transakcji w HTR w trybie DU a innych w trybie SM pozwala niekiedy na poprawę przepustowości i skalowalności systemu nawet o kilkadziesiąt procent. Wykorzystanie mechanizmów uczenia maszynowego w HTR pozwoliło na automatyczne i niemal natychmiastowe dostosowywanie się systemu do zmieniającego się obciążenia. Tym samym w opinii autora główna teza dysertacji została udowodniona.

Bibliografia

[1] C. H. Papadimitriou, "The serializability of concurrent database updates," *Journal of the ACM*, vol. 26, no. 4, 1979.

[2] B. Charron-Bost, F. Pedone, and A. Schiper, eds., *Replication - Theory and Practice*, vol. 5959 of *Lecture Notes in Computer Science*. 2010.

[3] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM (CACM)*, vol. 21, July 1978.

[4] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, Dec. 1990.

[5] F. B. Schneider, "Synchronization in distributed programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, Apr. 1982.

[6] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser, "Managing update conflicts in Bayou, a weakly connected replicated storage system," in *Proceeding of SOSP '95: the 15th ACM Symposium on Operating Systems Principles*, Dec. 1995.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Operating Systems Review*, vol. 41, Oct. 2007.

[8] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Operating Systems Review*, vol. 44, Apr. 2010.

[9] D. Abadi, "Consistency tradeoffs in modern distributed database system design: CAP is only part of the story," *Computer*, vol. 45, Feb. 2012.

[10] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao,

L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally-distributed database," in *Proceedings of OSDI '12: the 10th USENIX Conference on Operating Systems Design and Implementation*, Oct. 2012.

[11] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte, "F1: A distributed SQL database that scales," *Proceeding of Very Large Data Base (VLDB) Endowment*, vol. 6, Aug. 2013.

[12] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in *Proceedings of ISCA '93: the 20th International Symposium on Computer Architecture*, June 1993.

[13] P. Romano, N. Carvalho, and L. Rodrigues, "Towards distributed software transactional memory systems," in *Proceedings of LADIS '08: the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, Sept. 2008.

[14] J. Cachopo and A. Rito-Silva, "Versioned boxes as the basis for memory transactions," in *Proceedings of OOPSLA '05: Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, Oct. 2005.

[15] G. Korl, N. Shavit, and P. Felber, "Noninvasive concurrency with Java STM," in *Proceedings of MULTIPROG '10: the 3rd Workshop on Programmability Issues for Multi-Core Computers*, Jan. 2010.

[16] P. A., Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Addison-Wesley, 1987.

[17] R. Guerraoui and M. Kapalka, *Principles of Transactional Memory*. Synthesis Lectures on Distributed Computing Theory, 2010.

[18] H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky, "A programming language perspective on transactional memory consistency," in *Proceedings of PODC '13: the 32nd ACM Symposium on Principles of Distributed Computing*, June 2013.

[19] H. Attiya, S. Hans, P. Kuznetsov, and S. Ravi, "Safety of deferred update in transactional memory," in *Proceedings of ICDCS '13: the 33rd IEEE International Conference on Distributed Computing Systems*, July 2013.

[20] S. Doherty, L. Groves, V. Luchangco, and M. Moir, "Towards formally specifying and verifying transactional memory," *Formal Aspects of Computing*, vol. 25, no. 5, 2013.

[21] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, "Using optimistic atomic broadcast in transaction processing systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, July 2003.

[22] R. Palmieri, F. Quaglia, and P. Romano, "AGGRO: Boosting STM Replication via Aggressively Optimistic Transaction Processing," in *Proceedings of NCA 2010: the 9th IEEE International Symposium on Network Computing and Applications*, Feb. 2010.

[23] D. Sciascia, F. Pedone, and F. Junqueira, "Scalable deferred update replication," in *Proceedings of DSN '12: the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2012.

[24] D. Sciascia and F. Pedone, "RAM-DUR: In-Memory Deferred Update Replication.," in *Proceedings of SRDS '12: the 31st IEEE International Symposium on Reliable Distributed Systems*, Oct. 2012.

[25] B. Kemme and G. Alonso, "Don't be lazy, be consistent: Postgres-R, a new way to implement database replication," in *Proceedings of VLDB 2000: the 26th International Conference on Very Large Data Bases*, Sept. 2000.

[26] M. Couceiro, P. Romano, and L. Rodrigues, "PolyCert: Polymorphic self-optimizing replication for in-memory transactional grids," in *Proceedings of Middleware '11: the 12th ACM/IFIP/USENIX International Conference on Middleware*, Dec. 2011.

[27] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, 1990.

[28] R. Hansdah and L. Patnaik, "Update serializability in locking," in *Proceedings of ICDT '86: the 1st International Conference on Database Theory*, Sept. 1986.

[29] A. Adya, *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. Ph.D., MIT, 1999. Also as Technical Report MIT/LCS/TR-786.

[30] D. Imbs, J. R. G. De Mendivil Moreno, and M. Raynal, "On the consistency conditions of transactional memories," Research Report PI 1917, Inria, 2008.

[31] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys*, vol. 36, no. 4, 2004.

[32] H. Robbins, "Some aspects of the sequential design of experiments," *Bulletin of the American Mathematical Society*, vol. 58, no. 5, 1952.

[33] V. Kuleshov and D. Precup, "Algorithms for multi-armed bandit problems," *Computing Research Repository*, vol. abs/1402.6028, 2014.