Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Michał Wełnicki
Nr albumu: 189448

Biblioteka do testowania systemów rozproszonych z symulowaniem awarii

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr Janiny Mincer-Daszkiewicz
Instytut Informatyki

Wrzesień 2005

Oświadczenie kierującego pracą

Oświadczam, że niniejsza praca została przygotowana pod moim kierunkiem i stwierdzam, że spełnia ona warunki do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora (autorów) pracy

Streszczenie

W pracy przedstawiam bibliotekę wspomagającą testowanie systemów rozproszonych odpornych na awarie. W systemach takich zakłada się możliwość uszkodzenia pewnej części współpracujących komputerów lub łączącej je sieci z zachowaniem poprawnego funkcjonowania systemu.

Ponieważ spodziewane błędy zdarzają się rzadko i są trudne do przewidzenia, istotną część testów stanowi symulowanie awarii. Opisuję różne sposoby „wstrzykiwania" błędów, porównując je pod kątem oferowanej wierności symulacji, wpływu na wydajność testowanego systemu i stopnia kontroli nad nim. Pokrótce przedstawiam także niektóre istniejące narzędzia o podobnej funkcjonalności, wraz z opisem stosowanych przez nie technik.

Słowa kluczowe

system rozproszony, testy, symulacja awarii, wstrzykiwanie usterek, komunikacja sieciowa

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

D. Software
D.127. Testing and Debugging
D.127.6. Testing tools

Spis treści

Rozdział 1

Wprowadzenie

Motywacją dla powstania pracy był konkretny problem przygotowania testów dla pewnej aplikacji rozproszonej, nad którą pracuję dla firmy NEC Labs.

1.1. Opis testowanej aplikacji

Na testowany system składa się wiele serwerów komunikujących się w sieci między sobą oraz z procesami klienckimi. Serwery odczytują i zapisują na lokalnych dyskach dane dostarczone przez procesy klienckie.

W systemie może być wiele serwerów — jest on w założeniu skalowalny.

W docelowym systemie, dla maksymalnej wydajności, na jednym komputerze będzie działał pojedynczy proces serwera. Natomiast dla celów rozwoju i testowania przewidziano możliwość uruchomienia kilku serwerów na jednym komputerze, różnicując m.in. port używany do komunikacji oraz wykorzystywaną partycję na dane.

Serwery komunikują się w sieci używając własnego, niestandardowego protokołu. Obecnie protokół ten korzysta z TCP/IP, w przyszłości być może zostanie wykorzystany UDP. Każdy serwer może komunikować się bezpośrednio z dowolnym innym — oznacza to, że wszystkie komputery należące do systemu są bezpośrednio adresowalne (na poziomie IP). Natomiast na ogół każdy serwer komunikuje się tylko z niewielkim, w miarę stałym podzbiorem całego systemu (tzw. sąsiadami).

Podstawową cechą systemu jest jego odporność na awarie. System powinien zagwarantować poprawne funkcjonowanie w obliczu awarii pewnej liczby komputerów lub łączącej je sieci. Przewidywane rodzaje błędów to:

- zupełne zniszczenie pewnej liczby komputerów z serwerami,
- zatrzymanie serwera i ponowne jego uruchomienie po pewnym czasie (np. w wyniku awarii zasilania),
- czasowe zerwanie komunikacji pomiędzy pewnymi grupami serwerów,
- uszkodzenia pojedynczych dysków.

Serwery działają pod kontrolą systemu Linux, zostały napisane w C++. Procesy te są wielowątkowe i używają zaawansowanych mechanizmów zwiększających wydajność (m.in. asynchroniczne wejście–wyjście i bezpośredni dostęp do partycji dyskowych) oraz niezawodność (np. tworzenie dzienników).

Procesy serwerów są dość intensywne obliczeniowo, równocześnie wykazując dużą zależność od przepustowości systemu wejścia–wyjścia — zarówno sieci, jak i dysków.

1.2. Wymagania względem testów

Testom poddawany jest działający prototyp systemu. Powinny więc one przebiegać w warunkach możliwie zbliżonych do docelowych.

Zestaw testów uruchamiany będzie co noc, a także w czasie weekendów. Zależy nam na conocnym wykonywaniu większości testów, by szybko wykryć błędy popełnione przez programistów.

Testy uruchamiane będą na dedykowanych 2- oraz 4-procesorowych komputerach klasy PC, połączonych szybką, gigabitową siecią Ethernet. Liczba maszyn wahać się będzie w przedziale od kilku do kilkunastu. Dodatkowo dostępna jest niewielka liczba komputerów słabszych, przeznaczonych do uruchamiania procesów klienckich.

Aby przetestować odporność aplikacji zgodnie z przyjętym w niej modelem błędów, środowisko testów powinno umożliwiać wprowadzanie efektów następujących typów usterek:

- trwałe odłączenie wybranych węzłów z testowanego systemu: odpowiada zniszczeniu części komputerów,
- niskopoziomowy restart węzła: odpowiada awarii zasilania,
- zablokowanie, z możliwością późniejszego przywrócenia, komunikacji sieciowej między podanymi grupami węzłów.

Aby możliwe było testowanie wydajności systemu, mechanizmy generowania usterek nie powinny wprowadzać dużego narzutu. Najważniejsze, aby system działał z praktycznie pełną prędkością, gdy żadne awarie nie zostały jeszcze wprowadzone — środowisko testów nie powinno zaburzać pomiarów przepustowości systemu. Mniej istotne, ale także interesujące jest badanie wydajności systemu w momencie występowania awarii, np. podczas częściowego zerwania komunikacji.

Ponieważ obecnie serwery komunikują się po TCP, wiele aspektów typowych dla testowania protokołów komunikacyjnych, jak zamiana kolejności, opóźnianie oraz gubienie pojedynczych pakietów na poziomie warstwy łącza lub IP, nie jest istotne z punktu widzenia testów naszego systemu. Możemy założyć, że implementacja TCP w dzisiejszych jądrach systemu Linux była na tyle intensywnie przetestowana, że jest pozbawiona błędów, które bylibyśmy w stanie wykryć i poprawnie zdiagnozować naszymi testami.

Operacje, które chcielibyśmy wykonać, są na wyższym poziomie, np. ograniczenie przepustowości jakiegoś łącza lub jego całkowite zerwanie na jakiś czas. Natomiast manipulacje na pojedynczych pakietach nie będą interesujące dla testów tego systemu.

Mimo że testy dotyczą bardzo konkretnej aplikacji, pożądaną cechą biblioteki byłaby możliwość jej wykorzystania w przyszłych, podobnych projektach.

1.3. Cel pracy

Celem pracy jest zaprojektowanie oraz implementacja biblioteki wspomagającej testowanie systemów rozproszonych. Biblioteka ta musi umożliwiać symulowanie awarii opisanych w poprzednim punkcie, a więc:

- trwałego odłączenia wybranych węzłów z testowanego systemu,
- niskopoziomowego restartu węzła,
- zablokowania komunikacji sieciowej między podanymi grupami węzłów.

Stworzona biblioteka powinna, w miarę możliwości, spełniać wymagania opisane w punkcie 1.2.

1.4. Struktura pracy

W rozdziale 2 przedstawione zostały niektóre z istniejących narzędzi do testowania systemów rozproszonych. Każde z narzędzi zostało przeanalizowane pod kątem wykorzystania w systemie testów. Z przeprowadzonej analizy wynika potrzeba stworzenia nowego środowiska testów.

W rozdziale 3 opisuję różne sposoby symulowania usterek w testach. Przedstawiam wykorzystane mechanizmy systemu operacyjnego. Porównuję także cechy opisywanych metod, takie jak wydajność, realizm i łatwość użycia oraz implementacji.

Z kolei rozdział 4 zawiera krótki opis środowiska do automatyzacji zadań STAF, które służy jako podstawa zaimplementowanej biblioteki testującej.

Rozdział 5 poświęcony jest bibliotece, która powstała w ramach tej pracy. Opisany jest sposób instalacji, udostępniany interfejs, zaimplementowane techniki wprowadzania usterek (spośród tych opisanych w rozdziale 3) oraz sposób ich realizacji.

Rozdział 6 zawiera wyniki testów wydajności poszczególnych mechanizmów wprowadzania usterek.

Ostatni, 7 rozdział to podsumowanie pracy.

Rozdział 2

Przegląd istniejących narzędzi do testowania systemów sieciowych

W rozdziale tym opisane są istniejące środowiska do testowania systemów sieciowych. Wybrane zostały przykłady narzędzi działających na różnych zasadach. Wszystkie opisane narzędzia udostępniają jakieś mechanizmy symulowania usterek w systemach rozproszonych i ich użycie było rozważane podczas projektowania testów.

2.1. ns-2

Network Simulator[19, 8], w skrócie ns-2, jest to standardowe narzędzie do przeprowadzania symulacji sieci. W ramach symulacji, w pamięci symulatora tworzona jest reprezentacja topologii sieci i tych jej aktywnych elementów, które zostały wzięte pod uwagę przez twórcę eksperymentu.

Symulator ns-2 służy głównie do przeprowadzania eksperymentów z istniejącymi oraz projektowanymi protokołami i aplikacjami sieciowymi. Posiada bardzo rozbudowaną bibliotekę gotowych protokołów, urządzeń, a także aplikacji. Wykorzystywany był w wielu projektach badawczych, jest to więc narzędzie sprawdzone i dość rozpowszechnione.

Cechą wyróżniającą symulacji jest wirtualizacja upływu czasu oraz właściwie nieograniczona kontrola nad zdarzeniami zachodzącymi w środowisku.

Konfiguracja każdego eksperymentu (liczba węzłów, topologia sieci, działające źródła i ujścia danych) oraz przebieg symulacji definiowane są za pomocą skryptów w języku OTcl (obiektowej wersji Tcl). Skrypty te mogą korzystać z funkcji udostępnianych przez kontroler symulacji oraz moduły rozszerzające, napisane w C++.

2.1.1. Wady i zalety

ns-2 jest bardzo popularnym narzędziem do symulacji. Niewątpliwie sensownym pomysłem jest wykorzystanie go podczas projektowania systemu i przewidywania jego zachowania w dużych sieciach. Nie nadaje się on natomiast do testowania gotowego, działającego programu:

- większość elementów symulacji (np. implementacje protokołów, aplikacji) to przybliżone modele, zachowujące się z zewnątrz podobnie do rzeczywistych, ale często pozbawione ich funkcjonalności,
- podstawowa wersja ns-2 działa w jednym procesie, na jednym komputerze. Przy systemach o wysokich wymaganiach pamięciowych i obliczeniowych stanowi to problem dla

skalowalności eksperymentów.

2.2. UMLsim

Bardzo ciekawym projektem jest UML simulator (w skrócie UMLsim) [23, 24]. Jest to rozszerzenie User-mode Linux (UML) umożliwiające deterministyczną kontrolę nad upływem czasu widzianym zarówno przez jądro wirtualnego Linuksa, jak i procesy w nim uruchomione.

Dzięki wykorzystaniu User-mode Linux, symulacji poddać można gotowe, niezmienione programy, a także części jądra implementujące np. protokoły sieciowe.

Według autorów UMLsim ma być wykorzystywany głównie do rozwoju jądra Linuksa, poprzez testowanie nowych protokołów sieciowych, eksponowanie i dowodzenie istnienia warunków wyścigu procesów, wykonywanie testów regresji, a także szacowanie wydajności. Dzięki pełnej kontroli nad symulowanym systemem, możliwe jest doprowadzenie go do nietypowego i słabo przetestowanego stanu, którego osiągnięcie możliwe jest jedynie przy odpowiednim (mało prawdopodobnym) przeplocie równoległych operacji. UMLsim umożliwia więc efektywną analizę nietypowych ścieżek wykonania.

Scenariusze dla symulacji pisane są w specjalnym języku przypominającym C i Perl. Oparte są one na zdarzeniach i umożliwiają operacje typowe dla odpluskwiacza.

Mimo wykorzystania rzeczywistej, praktycznie niezmienionej implementacji systemu operacyjnego oraz uruchomionych aplikacji, UMLsim zaliczany jest do symulatorów. Spowodowane jest to właśnie wirtualizacją upływu czasu i deterministyczną kontrolą nad testowanym systemem.

UMLsim został z sukcesem użyty do analizy zachowania Linuksowej implementacji TCP na sieciach gigabitowych. Nie jest to jednak jeszcze produkt w pełni gotowy. Przede wszystkim wydajność symulacji jest bardzo niska (w porównaniu do tradycyjnych symulatorów jak ns-2 oraz do uruchomienia systemu na rzeczywistej maszynie). Częściowo spowodowane jest to zwiększoną wiernością symulacji w stosunku do np. ns-2, ale także ograniczeniami User-mode Linuksa i interfejsu do śledzenia `ptrace()`.

Mechanizmy udostępniane obecnie przez skrypty UMLsim są bardzo niskopoziomowe. Z tego powodu trudno jest kontrolować procesy uruchomione w wirtualnej maszynie. Co więcej, w obecnej wersji wirtualny węzeł może mieć tylko jedno połączenie sieciowe i to typu punkt do punktu. To ograniczenie właściwie dyskwalifikuje UMLsim jako środowisko testów naszego systemu.

2.2.1. Wady i zalety

Zalety:

- Używa rzeczywistego kodu aplikacji oraz jądra systemu operacyjnego (z niewielkimi modyfikacjami). W przeciwieństwie do symulacji abstrakcyjnych, nie ma niebezpieczeństwa, że model odbiega znacząco od rzeczywistej implementacji.
- Deterministyczna kontrola upływu czasu oznacza, że:
  - eksperymenty/przypadki testowe można powtarzać wielokrotnie, otrzymując te same efekty,
  - możliwe jest doprowadzenie testowanego systemu do mało prawdopodobnego stanu i sprawdzenie nietypowych ścieżek wykonania, przypadków brzegowych oraz szczególnie skomplikowanych.

Wady:

- Niska wydajność — testy trwałyby za długo, nie dałoby się testować przepustowości systemu.
- Obecna wersja jest ograniczona do jednego połączenia sieciowego na wirtualny węzeł.
- Programy uruchamiane są pod User-mode Linuksem, który nie działa idealnie.

2.3. UMLinux/FAUmachine

FAUmachine [5, 16] jest narzędziem do badania niezawodności systemów operacyjnych i ich odporności na awarie. Powstał w ramach projektu DBench, tworzącego narzędzia i metodologię do porównywania niezawodności systemów operacyjnych.

Głównym elementem FAUmachine jest maszyna wirtualna pozwalająca uruchomić kompletną, binarną instalację systemu operacyjnego pod kontrolą zewnętrznego Linuksa. Dzięki binarnej zgodności z rzeczywistą maszyną, możliwe jest uruchomienie dowolnej aplikacji w symulowanym środowisku. Drugą ważną częścią jest narzędzie symulujące przeróżne awarie wirtualnych komputerów i umożliwiające ocenę ich wpływu na wewnętrzne systemy operacyjne i aplikacje.

Początkowo FAUmachine był implementacją pomysłu podobnego do User-mode Linux, a więc umożliwiał uruchomienie lekko zmodyfikowanego jądra Linuksa jako procesu w trybie użytkownika. Tak jak w przypadku UML, konieczne było skompilowanie specjalnej wersji jądra, w której zmienione zostały trudne do zasymulowania instrukcje procesora. Oczywiście modyfikacji wymagało jedynie jądro, uruchamiane aplikacje nie wymagały żadnych zmian.

Jednak autorzy zauważyli, że konieczność przekompilowania jądra uniemożliwiała testowanie binarnych modułów oraz systemów operacyjnych, dla których kod źródłowy nie jest ogólnie dostępny. Dlatego implementacja FAUmachine została zmieniona i obecnie nie wymaga już żadnych zmian w jądrze systemu klienckiego. Możliwe jest już uruchomienie większości systemów opartych na Linuksie i OpenBSD, trwają też prace nad poprawieniem wierności emulacji tak, by można było uruchomić dowolne systemy, np. z rodziny Windows.

Z powodu tej dość istotnej zmiany projekt zmienił nazwę z pierwotnej UMLinux na FAUmachine i obecnie bliżej mu do produktów takich jak VMware i VirtualPC niż User-mode Linuksa.

Poza „typowymi" usterkami, takimi jak zerwanie komunikacji sieciowej i błędy urządzeń blokowych, FAUmachine symuluje także przekłamania w pamięci i rejestrach procesora. Co więcej, aby sztucznie nie zawężać możliwości wystąpienia tego typu usterek, symuluje je także w jądrze, a nie tylko aplikacjach symulowanego systemu. To wyklucza użycie bezpośrednio systemu operacyjnego gospodarza — błędy wstrzyknięte do prawdziwego jądra mogłyby spowodować utratę kontroli nad komputerem oraz procesem dokonującym symulacji i wynik działania mógłby nie zostać bezpiecznie zapisany.

Podstawową cechą wyróżniającą maszynę wirtualną UMLinux (a później FAUmachine) od User-mode Linuksa jest dążenie do minimalizacji zmian w symulowanym systemie, aby zbytnio nie zaburzyć wpływu awarii na system operacyjny i, pośrednio, uruchomione w nim aplikacje. User-mode Linux nie spełniał tego wymagania, gdyż:

- każdy proces symulowanego systemu działa jako oddzielny proces systemu gospodarza, a więc instrukcje wykonujące algorytm szeregowania oraz duża część kontekstu procesów są w pamięci jądra systemu gospodarza, niepodatnej na awarie,

- jądro systemu zostało rozbite na część w procesie nadzorującym i w procesie aplikacji. Wstrzykiwanie błędów do procesu nadzorcy może doprowadzić do propagacji błędnych zachowań do systemu zewnętrznego (skutkując np. nadpisaniem plików). Z drugiej strony, zupełne pominięcie nadzorcy sprawi, że większość implementacji jądra pozostanie nieprzetestowana.

Dlatego w UMLinux, w przeciwieństwie do User-mode Linux, jądro symulowanego systemu znajduje się w tej samej symulowanej pamięci co uruchomione pod nim aplikacje, a proces nadzorujący zajmuje się jedynie wirtualizacją wywołań funkcji systemowych i sygnałów oraz przekazywaniem ich do jądra wewnętrznego systemu.

2.3.1. Wprowadzanie błędów

FAUmachine umożliwia wprowadzanie następujących rodzajów błędów do wirtualnego systemu:

- pamięci RAM i rejestrów procesora — odwrócenie lub zablokowanie odpowiednich bitów,
- urządzeń blokowych — błędy odczytów/zapisów pojedynczych bloków lub całych urządzeń,
- komunikacji sieciowej — porzucanie wysyłanych/odbieranych pakietów z podanych interfejsów sieciowych,
- awarii zasilania — nagłe przerwanie działania wirtualnego systemu operacyjnego.

Każdy z wprowadzanych błędów może być *trwały*, *czasowy* lub *chwilowy*. Różnica między błędami chwilowymi a czasowymi odnosi się głównie do pamięci RAM i rejestrów procesora — błędy chwilowe oznaczają jednokrotne odwrócenie wartości wybranych bitów, natomiast trwałe lub czasowe symulują ich zablokowanie na jednej z wartości.

Błędy, które mają zostać wprowadzone do testowanego systemu, opisane są w pliku tekstowym zawierającym scenariusz wykonania. Każdy opis błędu zawiera jego typ, czas wprowadzenia, długość trwania i zakres działania (np. adresy błędnych komórek pamięci czy numery uszkodzonych sektorów).

Jak już wspomniałem powyżej, błędy dotyczą całego oprogramowania uruchomionego w maszynie wirtualnej, także jądra.

2.3.2. Implementacja maszyny wirtualnej

Aby zapewnić w miarę sensowną wydajność aplikacji uruchomionych w maszynie wirtualnej, program tych aplikacji musi być wykonywany przez rzeczywisty procesor, a nie symulator. Oznacza to, że do stworzenia środowiska wirtualnej maszyny w standardowym systemie operacyjnym konieczny jest mechanizm odnajdowania i przechwytywania:

- wywołań funkcji systemowych,
- prób wykonania instrukcji uprzywilejowanych,
- wyjątków zgłaszanych przez procesor np. w wyniku odwołania do niedostępnej strony.

W Linuksie służy do tego funkcjonalność śledzenia procesów `ptrace()`. Funkcja `ptrace()` używana jest m.in. przez odpluskwiacze takie jak *gdb*. Korzystający z `ptrace()` proces może podłączyć się do innego działającego procesu i sterować jego wykonaniem. Program śledzący może:

- być powiadamiany o wszystkich wywołaniach i powrotach z funkcji systemowych (parametr `PTRACE_SYSCALL`),
- przechwycić sygnały otrzymywane przez śledzony proces,
- odczytywać i modyfikować pamięć oraz rejestry procesora procesu śledzonego.

FAUmachine korzysta z `ptrace()` do śledzenia procesów działających w trybie użytkownika symulowanego systemu (śledzenie w ten sposób niezmienionego kodu jądra byłoby niepraktyczne, dlatego instrukcje jądra poddawane są najpierw przetłumaczeniu do postaci prostszej w śledzeniu). Wszelkie próby odwołań symulowanego procesu do jądra,czy to przez wywołanie funkcji systemowej, czy też spowodowanie wyjątku procesora, przechwytywane są przez proces nadzorujący. Przechwycone operacje są anulowane, a ich obsługą zajmuje się jądro działające w maszynie wirtualnej.

2.3.3. Symulacja błędów w FAUmachine

Maszyna wirtualna FAUmachine daje możliwość prostego wprowadzania błędów do symulowanego systemu. W kolejnych punktach opiszę krótko sposoby ich implementacji.

Błędy rejestrów procesora i pamięci

- Chwilowe: symulacja jest prosta, wystarczy nadpisać odpowiedni fragment pamięci FAUmachine lub zapamiętanego stanu rejestrów, korzystając z `ptrace(PTRACE_POKEDATA)` lub `ptrace(PTRACE_SETREGS)`.
- Trwałe/czasowe: symulacja jest mniej wydajna, bowiem przechwycić trzeba każdą próbę modyfikacji „uszkodzonej” komórki pamięci lub rejestru.
  - Pamięć: można zmienić atrybuty niektórych stron pamięci na „tylko do odczytu”.
  - Rejestry: za pomocą wykonania krokowego (bardzo powolne).

Błędy w symulowanych urządzeniach blokowych

Wszystkie odwołania do urządzeń i tak przechodzą przez proces nadzorujący. Wystarczy więc zmodyfikować program obsłujący wirtualne urządzenie tak, aby zwrócił komunikat o błędzie lub niepoprawne dane.

Zerwanie komunikacji sieciowej

Podobnie do urządzeń blokowych można rozwiązać symulację błędów sieci — wystarczy zignorować pakiety przekazywane przez writualny system operacyjny lub zmodyfikować ich zawartość.

Awaria zasilania

Z kolei do zasymulowania awarii zasilania wystarczy nagle zakończyć proces implementujący maszynę wirtualną, wysłając do niego sygnał `SIGKILL`.

Podsumowując, poza trwałymi uszkodzeniami pamięci/rejestrów, błędy wprowadzane przez maszynę wirtualną FAUmachine są proste w implementacji. Dość wiernie odwzorowują one rzeczywiste usterki i nie pogarszają wydajności wirtualnego systemu ponad narzut wprowadzony przez samą FAUmachine.

2.3.4. Wydajność

Niestety, już samo uruchomienie programów wewnątrz maszyny wirtualnej wprowadza dość duży narzut. Fakt, że symulowane błędy nie powodują dalszej nieefektywności niewiele zmienia.

Głównym źródłem nieefektywności FAUmachine jest użyty mechanizm `ptrace()`. Opisana powyżej metoda przechwytywania wywołań funkcji systemowych (`PTRACE_SYSCALL`) ma pewną wadę — rozpoczętej obsługi funkcji w jądrze systemu gospodarza nie można anulować. Oczywiście aplikacje uruchomione w wirtualnym systemie nie powinny bezpośrednio odwoływać się do zewnętrznego Linuksa. FAUmachine korzysta więc z pewnej sztuczki, polegającej na zamianie numeru wykonywanej funkcji systemowej, który zapisany jest w odpowiednim rejestrze na stosie procesu śledzonego. W efekcie, zamiast operacji zleconej przez proces, wywoływana jest nieszkodliwa funkcja systemowa, która nie powoduje żadnych efektów ubocznych — autorzy FAUmachine wybrali do tego celu `getpid()`.

Powyższy mechanizm działa, ale w jego wyniku z każdym wywołaniem funkcji jądra związanych jest wiele przełączeń kontekstu procesora:

1. Proces śledzony wywołuje funkcję systemową.
2. Jądro w kontekście procesu śledzonego budzi proces śledzący i wstrzymuje działanie śledzonego.
3. Proces śledzący podmienia numer funkcji na `getpid()`, przekazuje obsługę do wirtualnego jądra i wznawia wykonanie procesu śledzonego.
4. Jądro w kontekście procesu śledzonego wykonuje funkcję `getpid()` i ponownie wstrzymuje działanie.
5. Proces śledzący podmienia wartość powrotną w rejestrach procesu śledzonego na tę, którą otrzymał od wirtualnego jądra.
6. Proces śledzony kontynuuje działanie w trybie użytkownika.

Dla każdej funkcji systemowej konieczne są więc aż cztery przełączenia kontekstu między procesami, nie licząc przejść między trybem jądra i użytkownika.

Istnieje łata na jądro zewnętrznego Linuksa, udostępniająca poprawiony mechanizm przechwytywania funkcji systemowych. Umożliwia ona wywołanie funkcji systemowych jedynie instrukcjom znajdującym się w podanym zakresie adresów. Odwołania spoza dozwolonego zakresu powodują dostarczenie do procesu sygnału SIGINT. Dzięki temu udało się wyeliminować w ogole przełączenia kontekstu. Niestety, aby skorzystać z tego mechanizmu trzeba uruchomić własne, niestandardowe jądro na rzeczywistej maszynie.

Kolejnym potencjalnym źródłem narzutów jest dostęp do przestrzeni adresowej w zakresie 3 – 4 GB. Przestrzeń ta jest w Linuksie zarezerwowana dla jądra i procesy nie mogą z niej korzystać. Oczywiście jeśli w wirtualnej maszynie uruchomione jest standardowe jądro, to korzysta ono z tej właśnie przestrzeni. W efekcie wszystkie odwołania do pamięci jądra muszą

być symulowane, co pogarsza wydajność o rząd wielkości. Na szczęście twórcy FAUmachine udostępniają wersje Linuksa korzystające z innego adresu bazowego (2 GB zamiast 3 GB). Trzeba jednak pamiętać, by w wirtualnym systemie uruchamiać zmodyfikowane wersje jądra.

Autorzy FAUmachine przeprowadzili testy wydajności systemu uruchomionego w maszynie wirtualnej. Jako wyznacznik wydajności wykorzystali czas potrzebny do skompilowania jądra Linuksa. Jest to dość dobry scenariusz testu, korzystający intensywnie zarówno z systemu wejścia–wyjścia, jak i dokonujący pracochłonnych obliczeń (odpowiednio: czytając źródła/zapisując wyniki oraz generując skompilowaną postać programu).

Wyniki testu pochodzą z pracy [1]. Określają czas potrzebny do skompilowania jądra 2.4.18 na procesorze AMD Athlon XP 2100+ z 1 GB pamięci, dla różnych trybów wirtualizacji i na rzeczywistym systemie:

System	Czas kompilacji
Rzeczywista maszyna	141s
FAUmachine z `ptrace()` oraz zwyczajnym jądrem	1547s
FAUmachine z `ptrace()` oraz zmodyfikowanym jądrem	1439s
FAUmachine z SIGINT oraz zwyczajnym jądrem	635s
FAUmachine z SIGINT oraz zmodyfikowanym jądrem	514s

SIGINT oznacza poprawiony mechanizm przechwytywania wywołań jądra, nie wymagający przełączeń kontekstu.

Jak wynika z powyższego testu, nawet najlepsza wersja FAUmachine jest ciągle 3,5 raza wolniejsza niż rzeczywisty komputer. Dlatego, mimo dobrej implementacji błędów, FAUmachine nie spełnia wymagań opisanych w rozdziale 1.2.

2.3.5. Wady i zalety

Zalety:

- dokładna symulacja,
- testowanie systemu jako całości, umożliwiające wykrycie błędnego użycia interfejsów systemowych,
- prosta i w miarę dokładna symulacja awarii zasilania,
- możliwość bezkonfliktowego uruchomienia kilku serwerów na jednej fizycznej maszynie, gdy wydajność nie jest istotna.

Wady:

- symulacja jest zbyt powolna aby mogła być zastosowana w aplikacji intensywnie korzystającej z funkcji wejścia–wyjścia, szczególnie do analizy wydajności systemu podczas występowania błędów.

2.4. Podsumowanie

Poza opisanymi w tym rozdziale, istnieje wiele innych narzędzi do symulowania usterek i testowania systemów rozproszonych. Odnośniki do kilku z nich znaleźć można w bibliografii: [10, 3, 2, 4].

Żadne z analizowanych narzędzi do wprowadzania usterek nie spełnia w całości wymagań względem projektowanych testów. Niektóre narzędzia, jak ns-2, stworzone zostały z myślą o projektowaniu systemu i nie nadają się do testowania gotowych aplikacji. Z kolei inne, jak FAUmachine, mimo znakomitej realizacji usterek, nie spełniają wymagań co do wydajności środowiska testów.

Dlatego w ramach pracy stworzyłem własną bibliotekę symulującą usterki, która łączy dość wysoki realizm z niskim narzutem na wydajność testowanego systemu.

Rozdział 3

Techniki wprowadzania awarii

Ponieważ żadne z istniejących narzędzi nie okazało się odpowiednie do postawionego zadania, autor postanowił napisać własną bibliotekę testującą. W tym rozdziale opisane zostały różne techniki wprowadzania awarii do środowiska testów, które mogłyby zostać użyte w tworzonej bibliotece. Przedstawiona została krótka analiza wierności symulacji, niezawodności, prostoty implementacji oraz wydajności tych technik.

Jak już zostało opisane w poprzednich rozdziałach, żaden z analizowanych symulatorów nie nadaje się do testowania aplikacji przedstawionej w rozdziale 1.1. Pełne symulatory, wymagające stworzenia abstrakcyjnego modelu aplikacji, nie gwarantują całkowitej zgodności modelu z testowanym programem. Z kolei symulatory takie jak UMLsim, integrujące części rzeczywistych systemów z syntetycznym środowiskiem, nie spełniają wymagań wydajnościowych. Chociaż do mierzenia wydajności systemu można by użyć czasu wirtualnego, to czas wykonywania testów ograniczyłby znacznie ich liczbę w ciągu jednej nocy. Dlatego nie będziemy się już dalej zajmować symulatorami, a jedynie technikami wprowadzania usterek w rzeczywistym systemie.

3.1. Fizyczne operacje na sprzęcie

Najbardziej realistyczne i kompletne testy można by otrzymać wymuszając powstanie spodziewanych usterek, zamiast je symulować. Jak łatwo zauważyć, wymaganym awariom odpowiadają dość proste operacje na sprzęcie:

- wyłączenie zasilania,
- mechaniczne uszkodzenie lub odłączenie dysku,
- fizyczne zerwanie połączeń sieciowych, np. poprzez odłączenie kabli połączeniowych od kart sieciowych.

Niestety, aby mogły być wykonywane automatycznie, operacje te wymagałyby specjalistycznego sprzętu. Co więcej, częste wykonywanie powyższych czynności znacznie skróciłoby czas życia komputerów — np. typowe dyski nie są zaprojektowane z myślą o częstym włączaniu i wyłączaniu zasilania.

Opisane akcje mogą być sensownie wykorzystane w ograniczonym zakresie do testów wykonywanych ręcznie. Dobrym pomysłem byłoby chociaż jednokrotne potwierdzenie skuteczności aplikacji w reakcji na rzeczywiste awarie, a nie ich symulacje. Przykładowym błędem, praktycznie niewykrywalnym w inny sposób, mogą być niedociągnięcia użytego sprzętu lub sterowników.

Ponieważ jednak testy takie trudno zautomatyzować, w dalszej części przedstawione zostaną techniki czysto programowe.

3.2. Modyfikacja kodu źródłowego programu

Najprostszym sposobem wstrzykiwania błędów do aplikacji wydaje się być wprowadzenie odpowiednich modyfikacji do kodu źródłowego programu. Rozważmy przypadek blokowania komunikacji sieciowej: wszystkie miejsca w kodzie wysyłające bądź odbierające dane z sieci należy obudować sprawdzeniem warunku blokowania:

```
void sendToNetwork(struct ConnectionInfo *conn, void *buf, size_t bufLen)
{
   if (!disableSending)
   {
       write(conn->socketFd, buf, bufLen);
   }
}
```

Oczywiście warunek można rozbudować sprawdzając np. adres docelowy gniazda, a także wprowadzić kolejkowanie wysyłanych danych, aby zasymulować opóźnienia.

Niewątpliwą zaletą tego podejścia jest właśnie możliwość bezpośredniego skorzystania ze stanu procesu oraz wpłynięcia na niego. W ten sposób można aplikację doprowadzić do sytuacji nietypowych, trudnych do uzyskania wyłącznie zewnętrznymi interfejsami, i przetestować jej zachowanie. Jednocześnie ta dobra cecha może być w pewnych sytuacjach wadą: po zmianach nie jest naprawdę testowany wyjściowy program, tylko jego mniej lub bardziej zmodyfikowana wersja. Wszelkie modyfikacje logiki programu w zależności od korzystania bądź nie z trybu testującego mogą spowodować trudne do zdiagnozowania błędy pojawiające się wyłącznie wtedy, gdy wyłączone są instrumenty do ich wykrywania.

Zaszycie w kodzie źródłowym instrukcji służących jedynie do emulacji błędów może skomplikować logikę aplikacji. Mniejsza czytelność kodu powoduje więcej okazji do popełnienia błędów, których liczbę należy minimalizować. Poza tym po zakończeniu testów nie chcielibyśmy zostawiać tych mechanizmów w wersji programu rozprowadzanej do klientów. Należałoby więc użyć np. makr preprocesora, jeszcze bardziej komplikując źródła. Innym sposobem mogłoby być utrzymywanie zestawu łat na bazową wersję programu, aplikowanych tylko do wersji testowej. Jednak wtedy potrzebna byłaby dodatkowa praca utrzymywania tych łat w zgodzie z najnowszą wersją programu.

Sukces bądź porażka podejścia z modyfikacją źródeł w dużej mierze zależy od architektury testowanej aplikacji. Jeśli miejsca styku ze światem zewnętrznym są dobrze określone, np. poprzez konsekwentne korzystanie z jednej biblioteki do komunikacji w całym programie, to można wprowadzić niezbędne zmiany jedynie w tej bibliotece.

W przeciwnym razie dużym problemem tego podejścia może być odszukanie wszystkich miejsc, w których zmiany są konieczne. Jeśli w jakimś miejscu modyfikacja nie zostanie dokonana, może się okazać, że:

- testy kończą się porażką, bo nie udało się osiągnąć stanu zakładanego przez scenariusz testu; jest to sytuacja stosunkowo niegroźna, bo błąd prawdopodobnie zostanie w końcu zauważony i naprawiony, chociaż kosztem cennego czasu programistów,

- testy kończą się sukcesem, tyle że przetestowany został inny (być może prostszy) przypadek niż opisany w scenariuszu. Może się okazać, że testowany program wcale nie zachowuje się poprawnie, a poprawne przejście testu jest czystym przypadkiem. Taka sytuacja jest szczególnie groźna, bowiem bez raportu o porażce testu programista może nigdy nie szukać błędu w, wydawałoby się, sprawdzonej części kodu.

Świadomość istnienia powyższych problemów, nawet jeśli zmienione zostały wszystkie istotne miejsca w kodzie, wpłynęłaby na niską wiarygodność testów — programiści nie dowierzaliby takim testom i analizując raport o błędzie musieliby dodatkowo brać pod uwagę niedoskonałość biblioteki. Skomplikowałoby to jeszcze bardziej i tak trudny proces szukania błędu.

Kolejny problem wynika z wyboru miejsca, w którym symuluje się awarię: ponieważ dane są filtrowane na poziomie procesu użytkownika, nie jest testowana poprawność użycia interfejsów jądra. Przykładowo, rzeczywistym błędem naszej aplikacji było użycie blokującego wywołania `write()` do wysyłania danych, bez skorzystania z funkcji `setsockopt()` zmniejszającej czas oczekiwania na odpowiedź przez protokół TCP. Podczas testów opartych na powyższej technice system zachowywał się prawidłowo; dopiero po fizycznym odłączeniu jednej z komunikujących się maszyn okazało się, że wszystkie pozostałe serwery zasnęły na kilkanaście minut w funkcji `write()`, blokując wszelką komunikację w systemie.

Jeśli usterka jest zaimplementowana poprzez proste obudowanie wywołań systemowych i nie zależy od logiki aplikacji, to wierność takiego rozwiązania pozostawia dużo do życzenia — korzystając z interfejsu gniazd nie da się zablokować połączenia i poczekać na jego wygaśnięcie; co najwyżej można je zamknąć lub ignorować dane.

Awarię należy raczej symulować na wysokim poziomie, np. w bibliotece z jasno zdefiniowanym interfejsem, ukrywającej dostęp do surowych deskryptorów. Można w ten sposób ukryć fakt, że nie jesteśmy z stanie poprawnie zasymulować zerwanego połączenia. Poza tym często mamy wtedy dostęp do wysokopoziomowych informacji o połączeniu i przesyłanych danych. Niestety także to podejście nie jest idealne — nie testujemy faktycznej reakcji biblioteki na awarię, lecz sztucznie wprowadzamy proces w stan, w jakim, według intencji programisty, powinien się znaleźć w jej wyniku. Jednak celem testów jest właśnie zweryfikowanie założeń programisty i wykrycie subtelnych interakcji między elementami programu. Nawet jeśli zarówno biblioteka jak i kod jej używający przeszły swoje testy, to nie znaczy to jeszcze, że będą poprawnie współpracować (na tym polegają *testy integracji*).

Podsumowując, rozwiązanie z modyfikacją kodu źródłowego ma kilka zalet:

- prostota (przy dobrej modularyzacji),
- wprowadzane mogą być bardzo specyficzne błędy, np. zależne od wewnętrznego stanu procesu,
- na ogół niski narzut,

ale także kilka znaczących wad:

- nie zawsze jest możliwe do zastosowania — możemy nie mieć kodu źródłowego np. niektórych bibliotek,
- wprowadzanie specjalnych mechanizmów do testowania może negatywnie wpłynąć na czytelność kodu,
- dobre, wysokopoziomowe zmiany są na ogół specyficzne dla programu, trudno więc stworzyć wspólną bibliotekę — ta sama praca musi być wykonana dla każdego projektu,

- często nie ma gwarancji, że obsługa symulowania awarii została dodana we wszystkich miejscach, w których awaria może się objawić,
- nie jest testowana poprawność użycia interfejsów między procesem a systemem operacyjnym. Niektóre problemy mogą wynikać z interakcji z systemem operacyjnym, w tym przypadku nie zostaną wykryte,
- niski realizm, szczególnie z punktu widzenia węzła po drugiej stronie połączenia.

3.3. Przechwytywanie funkcji systemowych

Przedstawiona w poprzednim punkcie bezpośrednia modyfikacja kodu programu borykała się z problemami mieszania implementacji testów z logiką aplikacji oraz możliwością pominięcia niektórych odwołań do urządzeń zewnętrznych.

W tym punkcie przedstawię kilka metod wpływania na działanie gotowej aplikacji poprzez przechwycenie wywoływanych przez nią funkcji bibliotecznych i systemowych. Techniki te pozwalają w dużym stopniu ograniczyć lub całkiem wyeliminować powyższe problemy, kosztem operowania na niższym poziomie.

3.3.1. Mechanizmy przechwytywania wywołań funkcji

Zadanie jest następujące: mając daną gotową aplikację (w różnej formie: źródła, pliki obiektowe, plik wykonywalny) i własne implementacje niektórych funkcji systemowych, chcielibyśmy zmusić proces do wywoływania naszych funkcji zamiast rzeczywistych odwołań do systemu operacyjnego.

Okazuje się, że efekt ten da się osiągnąć bez ingerencji w jądro Linuksa — istnieją mechanizmy działające całkowicie w trybie użytkownika, a pozwalające na modyfikacje styku procesu z systemem operacyjnym.

Podczas konsolidacji: opakowywanie symboli

Konsolidator to program łączący pliki obiektowe powstałe po skompilowaniu poszczególnych jednostek kompilacji do pojedynczego pliku wykonywalnego. Głównym zadaniem konsolidatora jest złączenie odpowiednich sekcji poszczególnych plików oraz zamiana symbolicznych odwołań do nazw funkcji (np. `getpid`) na adresy względne w wynikowym programie. Adresy symboli, które nie zostały zdefiniowane w żadnym z plików składowych, a są eksportowane przez bibliotekę dynamiczną, zostają zamienione na adresy dopiero w momencie ładowania programu przez konsolidator dynamiczny (ld.so).

Konsolidator GNU ld [13], używany na systemach Linux, ma specjalną opcję służącą do podmieniania implementacji niektórych funkcji. Gdy do ld podana zostanie opcja `--wrap nazwa`, wszystkie odwołania do symbolu `nazwa` zostaną zmienione na odwołania do `__wrap_nazwa`, zaś odwołania do `__real_nazwa` na samo `nazwa`. Wówczas skonsolidowana aplikacja zamiast wywoływać funkcję `nazwa()` będzie wywoływała funkcję `__wrap_nazwa()`, która musi być zdefiniowana w jednym z dołączanych plików.

Mechanizm ten jest powszechnie stosowany m.in. przez narzędzia wykrywające błędy w zarządzaniu pamięcią przez programy w C, np. ElectricFence. Na ogół narzędzia te przychwytują funkcje `malloc()` i `free()`, dodając własne sprawdzenia i modyfikując zachowanie oryginału.

Wydajność takiego rozwiązania jest bardzo dobra — jest to po prostu wywołanie lokalnej, statycznie dołączonej funkcji w procesie.

Zaletą techniki opakowywania symboli jest to, że działa także dla programów złączonych statycznie.

Przedstawiony sposób przechwytywania funkcji nie jest całkowicie niezawodny — aplikacja można obejść implementację opakowującą, np. wywołując bezpośrednio funkcje systemowe bez pośrednictwa standardowej biblioteki. Oczywiście większość aplikacji tego nie robi, warto jednak pamiętać, że nie jest to dobra technika analizy nieznanego programu.

Rozwiązanie to ma jednak pewną wadę — ingeruje w proces konsolidacji programu. Nie zadziała więc z gotowym plikiem wykonywalnym. Z punktu widzenia testowanej aplikacji oznacza to, że albo trzeba korzystać zawsze z wersji z podmienionymi symbolami, albo muszą być tworzone dwie wersje pliku wykonywalnego — czysta oraz z biblioteką testującą.

Bibilioteki dynamiczne: LD_PRELOAD

Jak zostało wspomniane przy opisie procesu konsolidacji, symbole, które są zdefiniowane w bibliotece dynamicznej pozostają niezwiązane aż do momentu uruchomienia programu. Odnajdowaniem i ładowaniem bibliotek dynamicznych oraz wiązaniem eksportowanych przez nie symboli zajmuje się dynamiczny konsolidator ld.so [14].

Istnieje szereg zmiennych środowiskowych sterujących zachowaniem ld.so. Najbardziej znaną spośród nich jest LD_LIBRARY_PATH, określająca listę dodatkowych katalogów, w których należy szukać bibliotek. Z punktu widzenia przechwytywania funkcji systemowych interesująca jest zmienna LD_PRELOAD: dynamiczny konsolidator w pierwszej kolejności wczytuje wszystkie biblioteki wskazane przez tą zmienną, nawet jeśli nie są one potrzebne do uruchomienia programu. Ponieważ ld.so traktuje symbole znalezione wcześniej jako ważniejsze, funkcje eksportowane z załadowanej biblioteki przykryją oryginalne funkcje o tych samych nazwach, zdefiniowane np. w standardowej bibliotece języka C.

Prosty przykład:

```
#include <stdio.h>
#include <sys/types.h>

pid_t getpid()
{
    printf("Proces wywołał getpid()!\n");
    return 17;
}
```

Kompilacja i użycie biblioteki wygląda następująco:

```
gcc -Wall -shared -fPIC getpidwrapper.c -o libgetpidwrapper.so
LD_PRELOAD=./libgetpidwrapper.so program
```

Pozostaje jeszcze problem wywołania oryginalnej funkcji z wewnątrz biblioteki opakowującej. Zwyczajne odwołanie do `getpid()` spowodowałoby nieskończoną rekursję. Stosuje się na ogół jedno z dwóch podejść:

- duplikowanie kodu z libc wywołującego odpowiednią funkcję systemową w jądrze, lub

- pobieranie adresu rzeczywistej implementacji w standardowej bibliotece, korzystając ze specyficznego dla systemów GNU interfejsu dynamicznego konsolidatora.

Przykład poprawiony:

```
#define _GNU_SOURCE
#include <dlfcn.h>
#include <stdio.h>
#include <stdlib.h>

#include <sys/types.h>
#include <unistd.h>

static pid_t (*real_getpid)() = 0;

static void getpidwrapper_init() __attribute__((constructor));

static void getpidwrapper_init()
{
    real_getpid = dlsym(RTLD_NEXT, "getpid");
    if (real_getpid == NULL)
    {
        fprintf(stderr, "Failed to dlsym() real getpid\n");
        exit(1);
    }
}

pid_t getpid()
{
    printf("Proces wywołuje getpid()!\n");

    pid_t retVal = real_getpid();

    printf("getpid zwrócił %d!\n", retVal);

    return retVal;
}
```

Użyta w przykładzie flaga `RTLD_NEXT` oznacza odnalezienie symbolu o podanej nazwie w następnej (w kolejności wyszukiwania symboli) bibliotece dynamicznej po tej, z której wywołana została funkcja `dlsym`. Dzięki temu odnaleźć można rzeczywistą implementację przechwyconej funkcji.

Funkcja `getpidwrapper_init()` zostanie wykonana w momencie załadowania biblioteki dynamicznej, ponieważ oznaczona jest jako *konstruktor*. Dzięki temu, że zmienne biblioteki są inicjowane w momencie jej ładowania, nie jest konieczne późniejsze synchronizowanie dostępu do nich w programie wielowątkowym.

Zalety:

- nie wymaga ingerencji w proces budowania, działa z dowolnym programem korzystającym z bibliotek dynamicznych,
- niewielki narzut — taki, jak na wywołanie funkcji eksportowanej przez bibliotekę dynamiczną,

Wady:

- nie działa dla plików złączonych statycznie,
- nie działa dla bezpośrednich wywołań funkcji systemowych z aplikacji,
- może nie działać dla wewnętrznych wywołań między funkcjami w tej samej bibliotece (np. użycie `write()` przez `printf()` może nie zostać przechwycone, jeśli adres oryginalnego symbolu był wykorzystany już podczas konsolidowania biblioteki libc),
- użycie RTLD_NEXT ogranicza przenośność do systemów opartych na dynamicznym konsolidatorze GNU.

Ptrace

Jak już zostało opisane w rozdziale 2.3.2, funkcja `ptrace(PTRACE_SYSCALL)` informuje proces śledzący o wszystkich funkcjach systemowych wywoływanych przez proces śledzony.

Użycie `ptrace()` rozwiązuje problem aplikacji odwołujących się bezpośrednio do jądra, bez pośrednictwa standardowej biblioteki. Jednak również ta technika ma swoje wady:

- Rozpoznanie numeru wywoływanej funkcji systemowej i przetwarzanie jej argumentów wymaga niskopoziomowej wiedzy o sposobie wołania funkcji systemowych i znaczeniu rejestrów w danej architekturze. Przeniesienie techniki na systemy inne niż Linux na procesorach x86 może być pracochłonne.
- Każde wywołanie funkcji systemowej dwukrotnie przełącza kontekst na proces śledzący i z powrotem — przed wejściem i po wyjściu z funkcji. Ponieważ dotyczy to wszystkich funkcji systemowych, narzut może być duży.

Problem niskiej wydajności jest w pewnym stopniu zmniejszony przez fakt, że śledzenie za pomocą `ptrace()` nie musi odbywać się cały czas — można je włączać jedynie wtedy, gdy jest potrzebne.

3.3.2. Symulacja awarii sieci — przechwytywanie wywołań `write()`

Jest to proste przeniesienie techniki opisanej w punkcie dotyczącym modyfikowania kodu źródłowego. Zamiast modyfikować kod aplikacji, wystarczy przechwycić wykonywane przez nią wywołania `write()`. Jeśli żądany deskryptor jest połączeniem sieciowym i adres docelowy (pobrany za pomocą funkcji `getpeername()`) powinien być zablokowany, to wywołanie `write()` jest ignorowane; w przeciwnym wypadku wywoływana jest oryginalna funkcja.

Ponieważ zmieniona funkcja `write()` przechwytuje zapisy do wszystkich deskryptorów, konieczne jest rozpoznanie, które z nich oznaczają gniazda i z jakim adresem docelowym są związane. Informacji tych dostarcza funkcja systemowa `getpeername()` — dla gniazd przekazuje adres drugiego końca połączenia (wraz z rodziną adresów, do jakiej to gniazdo należy), zaś dla innych deskryptorów przekazuje błąd `ENOTSOCK`. Przy każdym wywołaniu `write()`

trzeba więc wywołać dodatkową funkcję systemową. Narzut z tym związany nie powinien być bardzo duży, ewentualnie można zapamiętywać typ deskryptora raz sprawdzonego — oczywiście wtedy trzeba dodatkowo przechwycić `close()`.

Przechwytywanie wywołań `read()`/`write()` w libc jest nie do końca poprawne: istnieje dużo różnych funkcji wywołujących `write()` wewnętrznie, np. rodzina funkcji typu `printf()`. Takie wewnętrzne odwołania wewnątrz biblioteki nie muszą przechodzić przez dynamiczne wyszukiwanie symboli. Z kolei `ptrace()` powoduje już zauważalny narzut i to na wszystkie wywołania funkcji systemowych.

Poza tym jest wiele sposobów wysłania danych po połączeniu sieciowym, nawet na poziomie interfejsu jądra: `send()`, `sendto()`, `sendmsg()`, `sendfile()`, teoretycznie `mmap()`. Trzeba by więc uważać, żeby obsłużyć je wszystkie.

Największą wadą tej techniki jest to, że połączenie nie jest naprawdę zrywane. Ciężko jest więc przetestować poprawność użycia interfejsów jądra, np. ustawiania maksymalnego czasu oczekiwania połączenia. Można teoretycznie przechwytywać odpowiednie wywołania `setsockopt()` i przekazywać błąd dopiero po pewnym czasie, a wcześniej ignorować dane przychodzące i wychodzące po tym gnieździe. Powoduje to jednak znaczną komplikację biblioteki testującej, i właściwie duplikację kodu jądra w tej bibliotece. Co więcej, ponieważ te same osoby pisały testowany program, bibliotekę testującą zaimplementują zgodnie z własnym zrozumieniem API systemu. Nie zostanie zweryfikowane, czy funkcje systemowe rzeczywiście działają tak, jak się wydaje programiście.

Szczególnie z punktu widzenia zdalnego węzła, ten sposób nie odpowiada dokładnie usterce, którą próbuje symulować — mimo że dane są ignorowane przez proces, stos TCP w jądrze i tak odpowiada potwierdzeniami na pakiety przesyłane przez zdalny węzeł. Z punktu widzenia procesu na tamtym węźle, dane poprawnie trafiają do odbiorcy i połączenie nigdy nie zostanie rzeczywiście zerwane.

3.3.3. Zamykanie połączeń

Zamiast ignorować dane czytane i pisane do gniazda można symulować zerwanie połączenia poprzez zamknięcie go. Należałoby przechwytywać wywołania `socket()`/`connect()`/`listen()` procesu i w momencie awarii wykonać `close()` na deskryptorach połączeń z zablokowanymi węzłami. Oto właściwości tego rozwiązania:

- Narzut jest tylko na otwieranie połączeń (trzeba zapamiętać, które deskryptory oznaczają połączenia sieciowe).
- Można wykorzystać dowolną z metod przechwytywania wywołań, zarówno przez opakowywanie symboli, jak i przez `ptrace()`. Niewiele funkcji standardowej biblioteki wewnętrznie otwiera połączenia sieciowe, nie powinno więc być problemów z przechwytywaniem wywołań potrzebnych funkcji.
- Podobnie jak poprzednio, realizm usterki nie jest najlepszy: po zamknięciu deskryptora, druga strona natychmiast się o tym dowie. Symulacja nie uwzględnia więc długich czasów oczekiwania TCP na zagubione pakiety.

3.3.4. Podsumowanie

Prawdopodobnie wszystkie rozwiązania polegające wyłącznie na przechwytywaniu funkcji systemowych będą miały problemy z realizmem wprowadzanych awarii. Interfejs funkcji systemowych jest po prostu miejscem bardzo odległym od tego, w którym występują symulowane błędy. Dlatego zamiast prostych błędów, trzeba zasymulować reakcję na nie wielu warstw

przetwarzania w jądrze systemu, co jest skomplikowane i często poza zasięgiem zwykłego procesu.

Pewnym rozwiązaniem tego problemu mogłoby być zaimplementowanie części stosu TCP/IP w bibliotece przechwytującej funkcje systemowe i korzystanie z własnej implementacji zamiast tej z jądra. Byłoby to jednak zadanie bardzo pracochłonne.

3.4. Wirtualne maszyny

Najlepszą wierność przy programowej symulacji usterek można osiągnąć uruchamiając testowaną aplikację wewnątrz maszyny wirtualnej. Zarówno aplikacja, jak i system operacyjny uruchomione są w środowisku odwzorowującym rzeczywisty komputer. Jedynym sposobem komunikacji ze światem zewnętrznym dla uruchomionego w maszynie wirtualnej oprogramowania jest skorzystanie z symulacji sprzętu dostarczanej przez tę maszynę.

Interfejs udostępniany przez większość maszyn wirtualnych uruchomionym w nich systemom operacyjnym przypomina operacje na rzeczywistym sprzęcie. Jest to dość naturalne, zważywszy, że takiego właśnie interfejsu spodziewają się tradycyjne systemy operacyjne. Dzięki temu styk maszyny wirtualnej z uruchomionym w niej systemem operacyjnym jest doskonałym miejscem do wprowadzania usterek — wszelkie modyfikacje zachowania na tym poziomie uwzględniają całą drogę przetwarzania, od sterowników wirtualnych urządzeń aż do aplikacji.

Zaleta ta została już zauważona i wykorzystana w opisanej w rozdziale 2.3 FAUmachine. Niestety, implementacja stworzona w tamtym projekcie jest zbyt mało wydajna. Istnieje jednak więcej maszyn wirtualnych — co prawda nie zostały one stworzone z myślą o testach, ale można je w tym celu wykorzystać. Jakość oraz wydajność niektórych dostępnych maszyn wirtualnych opisana została w dalszej części tego punktu.

3.4.1. User-mode Linux

User-mode Linux [21] jest oryginalną implementacją pomysłu, na którym wzorowali się twórcy opisanego już wcześniej FAUmachine. Jest to jądro Linuksa zmienione tak, by działało jako zwykły proces użytkownika pod kontrolą systemu gospodarza.

Niestety, User-mode Linux ma podobne problemy z wydajnością co FAUmachine — także tutaj za podstawę działania służy funkcja `ptrace()`. Ponadto, w przeciwieństwie do FAUmachine, Linux dla architektury UML różni się niekiedy dość znacznie od implementacji dla prawdziwego sprzętu.

Główne wady to:

- słaba wydajność, szczególnie przy niezmienionym jądrze systemu gospodarza,
- brak wersji SMP,
- niestabilność.

3.4.2. XEN

XEN [22, 26] jest nietypową maszyną wirtualną, oferującą bardzo dobrą wydajność w porównaniu z tradycyjnymi rozwiązaniami.

XEN działa bezpośrednio na sprzęcie, ponad uruchomionymi systemami operacyjnymi. Pełni rolę warstwy pośredniczącej i nadzorującej, podobnej nieco do prostego systemu operacyjnego. Warstwa ta zapewnia izolację między uruchomionymi systemami operacyjnymi i zarządza dostępem do sprzętu.

Dobrą wydajność XEN zawdzięcza temu, że:

- Działa bezpośrednio na sprzęcie, pomijając narzuty wymuszone systemem operacyjnym gospodarza. Przykładowo, XEN nie musi rozwiązywać problemów wydajności mechanizmu `ptrace()`.
- Nie stara się emulować dokładnie architektury, na której jest uruchomiony, a jedynie wyidealizowany interfejs do sprzętu. Autorzy XEN nazywają tę technikę *parawirtualizacją*.

XEN zapewnia zgodność z rzeczywistą architekturą jedynie dla programów działających w trybie użytkownika. Każda aplikacja działająca na prawdziwym sprzęcie zadziała również pod XEN. Jest to możliwe dlatego, że większość architektur sprzętowych, w tym *x86*, daje możliwość efektywnego zaimplementowania wirtualizacji procesów nieuprzywilejowanych.

Najczęściej źródłem nieefektywności tradycyjnych maszyn wirtualnych jest symulowanie uprzywilejowanych instrukcji trybu jądra. Nie wszystkie cechy i instrukcje trybu jądra da się zasymulować w sposób wydajny, czasem konieczne jest interpretowanie bądź dynamiczne przekompilowywanie wykonywanych instrukcji.

XEN nie próbuje emulować środowiska trybu jądra. Systemy operacyjne uruchamiane są na niższym poziomie uprzywilejowania i nie mogą bezpośrednio korzystać z uprzywilejowanych instrukcji ani odwoływać się do sprzętu. W zamian udostępniany jest interfejs *hiperwywołań*, służących do komunikacji z warstwą zarządzającą — jest to mechanizm analogiczny do funkcji systemowych w zwykłych systemach operacyjnych, ale implementujący inną funkcjonalność.

Oczywiście parawirtualizacja oznacza, że systemy operacyjne muszą być specjalnie przenoszone na architekturę XEN. Na wirtualną platformę XEN przeniesiony został już m.in. Linux w wersji 2.6.

Pod XEN uruchomionych może być wiele systemów operacyjnych, ale pierwszy z nich jest wyróżniony — może on zarządzać pozostałymi maszynami wirtualnymi i zapewnia obsługę sprzętu.

Dobrze rozwiązany został problem izolacji poszczególnych systemów operacyjnych. Zasoby fizycznego komputera rozdzielane są sprawiedliwie, nie jest możliwe, aby jedna maszyna wirtualna zagłodziła lub poważnie spowolniła pozostałe.

Mimo wszystkich zalet, XEN ma kilka wad, które poważnie utrudniają jego użycie w bibliotece testującej:

- Wymagany jest surowy dostępu do sprzętu, XEN działa ponad wszystkimi systemami operacyjnymi, łącznie z tym wyróżnionym, i musi być zainstalowany na własnej partycji.
- Narzut, choć niewielki jak na maszynę wirtualną, jest jednak dostrzegalny.
- Nie można prosto wyłączyć maszyny wirtualnej i uruchomić aplikacji z pełną wydajnością na standardowym systemie. Jedynym sposobem pozbycia się narzutu XEN jest zrestartowanie komputera i załadowanie systemu z innej partycji.
- Pod XEN nie działają biblioteki standardowe oparte na NPTL (*Native Posix Threading Library*). Jest to wydajna implementacja wątków pod Linuksem, korzystająca z TLS (*Thread Local Storage* — przestrzeni danych prywatnych dla wątku). Typ odwołań do pamięci używany przez biblioteki NPTL musi być emulowany przez XEN, działa więc bardzo wolno, a często nie działa w ogóle.

  Do stabilnej pracy konieczne jest wyłączenie bibliotek NPTL lub skompilowanie zmodyfikowanej wersji standardowej biblioteki glibc. Aplikacje intensywnie korzystające z

wątków bez NPTL działają mniej wydajnie i nie do końca zgodnie ze standardem POSIX.

- W obecnej wersji XEN co prawda sam wykorzystuje maszyny wieloprocesorowe, ale nie jest możliwe uruchomienie wieloprocesorowej wersji systemów operacyjnych.

3.5. Mechanizmy kontroli interfejsów sieciowych

Ponieważ interfejs funkcji systemowych okazał się nieodpowiednim miejscem do symulowania awarii sieci, a korzystanie z wirtualizacji pociąga za sobą duży narzut, symulację tę trzeba przeprowadzić w jądrze systemu. Najczęstszym sposobem zmiany funkcjonalności jądra jest napisanie modułu, bądź, przy szerszych zmianach, stworzenie łaty na oryginalną wersję Linuksa implementującą wymaganą funkcjonalność. Dzięki dostępności i otwartości kodu źródłowego Linuksa jest to rozwiązanie absolutnie wykonalne i często sensowne. Co więcej, zmiany wykonywane na własny użytek, jeśli okażą się przydatne dla większej liczny użytkowników, mogą zostać włączone do oficjalnej dystrybucji i przyczynić do dalszego rozwoju Linuksa.

Jednak modyfikacja jądra zawsze wiąże się z pewnym ryzykiem. Błędy w kodzie działającym w trybie jądra powodują na ogół zatrzymanie całej maszyny, dlatego moduły muszą być pisane ze szczególną starannością. Poza tym konieczny jest nakład pracy na utrzymanie modułów lub łat w synchronizacji z nowszymi wersjami systemu.

Okazuje się, że do wielu zastosowań nie jest już konieczne modyfikowanie jądra, a jedynie skorzystanie z jednego z ogólnych interfejsów udostępnianych przez różne moduły procesom działającym w trybie użytkownika.

3.5.1. Netfilter/iptables

Netfilter/iptables[18] jest to standardowe środowisko do filtrowania (ściana ogniowa), translacji adresów (NAT) i ogólnego modyfikowania pakietów w Linuxie. Netfilter jest to zestaw punktów rozszerzeń w różnych fazach obsługi pakietów sieciowych przez jądro. Do tych punktów rozszerzeń mogą się podłączyć moduły jądra, wpływając na przechodzące przez nie pakiety. Z kolei iptables jest specjalną strukturą tablic służącą do definiowania zestawów reguł. W połączeniu z modułami analizującymi i modyfikującymi pakiety, reguły te określają, które pakiety zostaną obsłużone przez jądro i w jaki sposób.

Netfilter jest częścią jądra Linuksa już od wersji 2.4 i jest szeroko wykorzystywany w wielu instalacjach systemu. Dzięki temu jest to produkt dobrze sprawdzony i dojrzały.

Ponieważ operacje na pakietach odbywają się w trybie jądra, a pakiety podczas przechodzenia przez reguły nie są kopiowane, mechanizm filtrowania nie wprowadza dużego narzutu na ich obsługę.

Każdy pakiet przechodzi przez łańcuchy reguł w odpowiednich tablicach. Oddzielne reguły można tworzyć m.in. dla pakietów przychodzących, wychodzących i przechodzących przez system, w różnych stadiach ich przetwarzania. Reguły zorganizowane są w listy — mają określoną kolejność i sprawdzane są po kolei. Jeśli warunki opisane w regule pasują do pakietu, to wykonywana jest związana z tą regułą akcja. W przeciwnym razie pakiet przechodzi do następnej reguły, a jeśli dojdzie do końca listy wykonywana jest akcja domyślna. Możliwe akcje są także zaimplementowane w modułach jądra, a najczęściej używane to:

ACCEPT: pakiet zostaje zaakceptowany do przetworzenia przez następne części jądra,

DROP: pakiet zostaje porzucony — jest usuwany z pamięci bez obsłużenia przez protokoły sieciowe lub sterownik,

REJECT: pakiet zostaje zignorowany, a w odpowiedzi do nadawcy wysyłany jest komunikat ICMP z informacją o błędzie.

Praktycznie wszystkie elementy opisanego procesu są rozszerzalne za pomocą odpowiednich modułów jądra.

3.5.2. TUN/TAP

Universal TUN/TAP [25] jest to specjalny sterownik interfejsu sieciowego do Linuksa, który zamiast wysyłać pakiety przez rzeczywiste urządzenie, przekazuje je do obsługi procesowi trybu użytkownika. Analogicznie, dla pakietów przekazanych z trybu użytkownika przez zwykły proces, symuluje ich odebranie przez interfejs sieciowy.

Moduł TUN/TAP tworzy:

- interfejsy sieciowe `tunX` oraz `tapX`,
- urządzenie znakowe `/dev/tun`, służące do komunikacji z procesami oraz zarządzania interfejsami sieciowymi.

Różnica między interfejsami `tun` i `tap` polega na rodzaju przesyłanych pakietów: `tun` działa na pakietach warstwy IP, natomiast `tap` symuluje kompletną sieć lokalną i działa na ramkach protokołu *Ethernet*.

Interfejs TUN/TAP pozwala zaimplementować wirtualną sieć w procesie trybu użytkownika. Ma to pewne zalety — w zwykłym procesie obsługa takiej sieci może być znacznie bardziej skomplikowana i rozbudowana niż w jądrze. Przykładowo, dość kłopotliwe jest w kontekście jądra otwieranie połączeń TCP, natomiast jest to proste w procesach użytkownika. Zwykły proces nie musi się przejmować regułami synchronizacji jądra, dzięki czemu może być mniej skomplikowany. Ponadto, TUN/TAP został przeniesiony na wiele systemów operacyjnych, między innymi FreeBSD i Solaris. Napisanie przenośnego procesu nie stanowi dużego problemu, za to przenośny moduł do jądra jest właściwie niemożliwy.

Oczywiście przetwarzanie pakietów w trybie użytkownika wiąże się z dużo większym narzutem niż w trybie jądra — dla każdego pakietu przechodzącego przez interfejs `tun` lub `tap` musi nastąpić przełączenie kontekstu na proces obsługujący. Testy rzeczywistej wydajności tego mechanizmu przedstawione będą w dalszej części pracy.

3.5.3. Cel QUEUE w iptables oraz libipq

Innym sposobem przekazywania pakietów do trybu użytkownika jest cel QUEUE [18, 6]. Jest to rozszerzenie do iptables, które definiuje nowy cel dla pakietów, analogiczny do ACCEPT, REJECT i DROP. Pakiety, które pasują do reguły z celem QUEUE, zostają przekazane do zainteresowanego procesu poprzez specjalny mechanizm gniazda *Netlink*. Proces użytkownika dostaje dane pakietu i na ich podstawie musi zdecydować o dalszym jego losie — pakiet może zostać porzucony lub zaakceptowany. Dodatkowo, proces może zmodyfikować dane pakietu.

Do obsługi pakietów w procesie udostępniana jest biblioteka `libipq`, dostępna wraz ze źródłami iptables. Biblioteka ta ukrywa cały proces tworzenia gniazda *Netlink* i komunikacji z jądrem, udostępniając prosty interfejs do pobierania pakietów i ustawiania stanu ich akceptacji.

W przeciwieństwie do TUN/TAP, libipq nie można użyć bezpośrednio do symulowania sieci — pakietów przekazanych przez mechanizm QUEUE na jednym węźle nie da się przesłać na inny węzeł i tam zaakceptować. Konieczne jest użycie innego mechanizmu do zasymulowania odebrania pakietu na zdalnym węźle. Jest to natomiast dobry mechanizm do przechwytywania pakietów przechodzących przez jądro i pisania skomplikowanych reguł ich akceptacji.

Cel QUEUE jest dołączany do większości dystrybucji pakietu iptables, zarówno w jądrach 2.4, jak i 2.6.

3.6. Symulacja awarii sieci za pomocą mechanizmów jądra

W poprzednim punkcie opisane zostało kilka mechanizmów Linuksa umożliwiających kontrolę nad wygenerowanymi oraz otrzymanymi pakietami. Spróbuję teraz przedstawić kilka możliwości skorzystania z tych interfejsów do zasymulowania awarii sieci łączącej testowane procesy. Sprecyzuję także wymagania względem takiej symulacji.

3.6.1. Skorzystanie z iptables

Jak widać z wcześniejszego opisu, cel DROP w iptables dobrze nadaje się do symulowania zgubienia pakietów w sieci. Można więc w prosty sposób zasymulować awarię karty sieciowej komputera:

```
iptables -A INPUT -j DROP
iptables -A OUTPUT -j DROP
```

Polecenia te wstawiają na koniec łańcucha pakietów przychodzących (INPUT) i wychodzących (OUTPUT) z węzła regułę, która pasuje do każdego pakietu i powoduje jego porzucenie.

Symulacja jest bardzo realistyczna, dla protokołu IP sytuacja będzie wyglądała jak zerwanie łączności z siecią lokalną.

W rzeczywistym środowisku całkowite zablokowanie komunikacji z węzłem jest jednak niepraktyczne. Po pierwsze, blokada komunikacji dotyczy także procesu nadzorującego wykonanie testu, trudno jest więc zsynchronizować moment usunięcia blokady. Poza tym blokowana jest możliwość zdalnego zalogowania się na komputer; jeśli blokada nie zostałaby zdjęta (np. wskutek błędu w scenariuszu testu), jedynym sposobem przywrócenia działania komputera byłoby usunięcie powyższych reguł na fizycznej konsoli. Ponieważ komputery testujące często znajdują się w niedostępnych miejscach (np. w piwnicy), jest to sytuacja bardzo niewygodna.

Aby przezwyciężyć ten problem można skorzystać z reguły dopasowującej pakiety po adresie docelowym:

```
iptables -A OUTPUT --destination blokowany_adres_ip -j DROP
```

Po wstawieniu takiej reguły blokowana będzie (cała) komunikacja z węzłem o podanym adresie IP. Jeśli scenariusze testów będą przygotowane tak, aby komputer nadzorujący testy nigdy nie został zablokowany, kontrola nad maszynami testującymi nie zostanie utracona.

Powyższe reguły są znacznie lepsze, ale jeszcze nie doskonałe. Ponieważ na pojedynczym komputerze testującym pracować może kilku programistów naraz, istotną sprawą jest ich izolacja. Przykładowo, w czasie dnia programiści uruchamiają te scenariusze, które podczas

nocnych testów wykryły błędy w ich kodzie. Jeśli kilka scenariuszy uruchomionych zostanie równocześnie na tych samych maszynach, to awarie wymuszone w jednym teście będą widoczne we wszystkich pozostałych.

Szczęśliwie istnieje moduł do iptables dopasowujący jedynie te pakiety wychodzące, które powstały na rzecz aplikacji podanego użytkownika:

```
iptables -A OUTPUT -d blokowany_adres \
    -m owner --uid-owner uid_wlasciciela_aplikacji -j DROP
```

W tej wersji wszelkie reguły blokujące komunikację będą dotyczyły jedynie procesów podanego użytkownika (zapewne tego, który zlecił blokadę). Różni użytkownicy nie będą sobie wzajemnie przeszkadzać. Dodatkową zaletą jest to, że wszystkie serwisy, których właścicielami są użytkownicy systemowi, będą działać bez przeszkód. Dzięki temu możliwe będzie zalogowanie się na maszynę po SSH, nawet jeśli w bibliotece testującej pojawią się błędy i reguły blokujące nie zostaną automatycznie zdjęte.

Teoretycznie serwer mógłby wykorzystać któryś z lokalnych serwisów należących do innego użytkownika i odkryć, że docelowy węzeł faktycznie jest osiągalny — przykładem takiego działania byłoby uruchomienie programu `ping`, który domyślnie ma ustawioną flagę `SUID` i wykonuje się na prawach użytkownika systemowego `root`. W praktyce nie ma to większego znaczenia, bowiem serwery na ogół nie polegają na zewnętrznych programach do ustalenia osiągalności (bądź nie) węzłów, z którymi chcą się komunikować.

3.6.2. Wiele procesów na jednej maszynie

Niektóre scenariusze testowe wymagają uruchomienia wielu procesów — ich liczba może przekraczać liczbę dostępnych komputerów. Poza tym czasami komputery testujące ulegają rzeczywistym usterkom. Gdyby testowane procesy były przypisane do komputerów na stałe, wszystkie testy korzystające z niedostępnego węzła zakończyłyby się porażką.

Wobec powyższych faktów przydatną cechą systemu testów okazuje się swoboda przydziału testowanych procesów do dostępnych maszyn testujących, a w szczególności możliwość uruchomienia kilku procesów na jednej maszynie.

Oczywiście, taka konfiguracja negatywnie wpływa na wydajność testowanej aplikacji, ponieważ procesy konkurują między sobą o dostęp do zasobów. W praktyce okazuje się jednak, że jest wiele scenariuszy, w których taki rozkład procesów nie jest przeszkodą:

- Równocześnie tylko kilka procesów aktywnie korzysta z zasobów, pozostałe są w stanie pasywnym (np. serwer zapasowy oczekujący jedynie na awarię serwera głównego).
- Niektóre procesy same wykorzystują niewiele zasobów, natomiast generują obciążenie dla innych procesów (przykładem są procesy klienckie) lub są konieczne do działania systemu (np. nadzorca testu).

Kolejną sytuacją, w której uruchomienie wielu procesów na jednym komputerze jest sensowne jest wspomniana już awaria komputera testującego. Na ogół podczas nocnych testów wszystkie dostępne komputery są w wykorzystywane, nie ma więc zapasowego węzła, który mógłby przejąć funkcję uszkodzonego. Bez wsparcia dla wielu procesów na węźle, cały zestaw testów nie mógłby się wykonać. Lepiej jest więc przenieść procesy z węzła uszkodzonego na jeden ze sprawnych i poznać wyniki testów poprawności (ignorując zaburzoną wydajność), niż nie uruchomić testów w ogóle.

Sposoby blokowania komunikacji opisane w punkcie 3.6.1 nie pozwalają niestety niezależnie blokować procesów uruchomionych na tym samym węźle. W następnych punktach przedstawione zostaną różne pomysły rozwiązania tego problemu. Do ich analizy zdefiniować można kilka cech pożądanych w mechanizmach blokowania komunikacji:

- Izolacja poszczególnych użytkowników (idealnie poszczególnych scenariuszy testowych) — blokady utworzone w jednym teście nie powinny wpływać na procesy należące do innego testu.

  Zasadność tej cechy przedstawiona już została w punkcie 3.6.1 (przy opisie dopasowania `uid-owner`).

- Procesy nie należące do żadnego testu nie powinny być blokowane.

- Możliwość niezależnego blokowania procesów uruchomionych na tym samym węźle.

- Możliwość zablokowania obu kierunków komunikacji na pojedynczym węźle — operacja zablokowania komunikacji między pojedynczym procesem a resztą systemu powinna być wykonywana lokalnie na blokowanym węźle, bez konieczności komunikacji z wszystkimi węzłami sieci.

  Większość definiowanych blokad komunikacji powoduje odłączenie małej grupy procesów od systemu (często jest to wręcz pojedynczy proces). Dobrze więc, żeby takie blokady były zaimplementowane wydajnie.

3.6.3. Przyporządkowanie pakietów do procesów: `pid-owner`

Do standardowej dystrybucji iptables dołączony jest moduł dopasowujący pakiety według procesu, na rzecz którego zostały one wygenerowane. Dopasowanie może następować po identyfikatorze procesu (`pid-owner`), sesji (`sid-owner`) lub poleceniu użytym do uruchomienia procesu (`cmd-owner`).

Moduł ten wydaje się być dobrym narzędziem do blokowania komunikacji pojedynczych procesów. Aby porzucać pakiety należące do procesu o danym identyfikatorze *pid*, wystarczy wstawić następującą regułę:

```
iptables -A OUTPUT -d blokowany_adres -m owner --pid-owner pid_procesu -j DROP
```

Niestety, w powyższym module odkryto błąd w synchronizacji dostępu do struktur jądra. Użycie modułu na systemie wieloprocesorowym może doprowadzić do blokady i w efekcie zatrzymania całego systemu operacyjnego. Dlatego od wersji 2.6.8 Linuksa dopasowanie po *pid*, *sid* i poleceniu zostało zabronione w maszynach wieloprocesorowych.

Błąd w synchronizacji okazał się na tyle poważny, że autorzy iptables uznali jego naprawienie bez znacznych ingerencji w jądro Linuksa za niemożliwe.

3.6.4. Alternatywy dla `pid-owner`

Ponieważ problem w implementacji `pid-owner` raczej nie zostanie szybko naprawiony, a komputery używane do testowania aplikacji są wieloprocesorowe, konieczne jest zasymulowanie działania `pid-owner` innym modułem.

Przechwytywanie numerów portów

Pierwszy ze sposobów polega na blokowaniu tych portów, które są w danej chwili otwarte przez testowany proces. Aby stwierdzić, które to porty, można przechwytywać próby otwarcia portów przez testowany proces. Można w tym celu użyć dowolnej z opisanych wcześniej technik przechwytywania funkcji systemowych.

Port TCP lub UDP przydzielany jest procesowi w wyniku wywołania `bind()` lub próby otworzenia połączenia (`listen()`, `connect()`, `sendto()`) na gnieździe dotychczas niezwiązanym z żadnym portem. Funkcja `bind()` służy do związania gniazda z lokalnym adresem IP i numerem portu. Gdy zostanie wywołana z zerowym numerem portu, port jest wybierany automatycznie przez jądro. Przydzielony numer można później odczytać korzystając z funkcji systemowej `getsockname()`.

Biblioteka opakowująca wywołania systemowe może więc działać według następującego algorytmu:

- jeśli wywoływana jest funkcja `bind()`:
  1. wywołać oryginalną funkcję `bind()`,
  2. pobrać przypisany adres i numer portu wywołując `getsockname()` na tym samym deskryptorze,
  3. poinformować o używanym porcie proces nadzorujący;
- jeśli wywoływana jest funkcja wykonująca automatyczne przypisanie portu (`listen()`, `connect()`, `sendto()`, `sendmsg()`):
  1. wykonać `getsockname()`,
  2. jeśli gniazdu nie jest przypisany żaden port (wartość przekazana przez `getsockname()` to 0), wykonać symulowaną funkcję `bind()` dla portu 0; spowoduje to automatyczne ustalenie portu i przekazanie informacji do nadzorcy,
  3. wywołać oryginalną funkcję.

Warto zauważyć, że nie ma tu wyścigu pomiędzy otwieraniem portu a zablokowaniem połączenia przez proces nadzorujący, jeśli gniazdo korzysta z protokołu TCP. Gniazdo związane z portem, ale jeszcze nie otwarte wywołaniami `listen()` lub `connect()` nie przyjmuje przychodzących pakietów. Niestety, w przypadku UDP sam proces przydzielenia portu otwiera możliwość komunikacji z gniazdem. W tym przypadku możliwe jest odebranie kilku pakietów na nowo otwartym gnieździe UDP pomimo trwającej blokady komunikacji.

Kolejnym problemem jest wykrycie momentu zamknięcia gniazda. Problem jest trudniejszy, ponieważ otworzyć port może tylko kilka wyspecjalizowanych funkcji systemowych, natomiast zamknięcie podlega normalnym regułom zamykania deskryptorów — mogą istnieć procesy potomne, które odziedziczyły gniazdo, gniazda są zamykane w momencie wykonywania `exec()` itd. Poprawne obsłużenie wszystkich tych przypadków byłoby trudne.

Dopasowanie po identyfikatorze grupy (GID)

Poza dopasowywaniem po PID i UID, pakiety w iptables można dopasowywać po identyfikatorze grupy, do której należy nadawca pakietu. Dopasowywanie po GID działa bez problemu na maszynach wieloprocesorowych.

Można wykorzystać ten mechanizm tworząc specjalne grupy dla potrzeb testów. Procesy, które mają być rozróżnialne przez iptables muszą być uruchomione z innymi identyfikatorami grup.

Zalety:

- działa na SMP,
- efektywne i pewne dopasowanie.

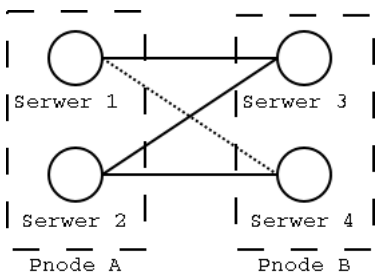
Wady:

- konieczność zdefiniowania specjalnej grupy,
- potrzebny jest program opakowujący potrafiący uruchomić testowany proces ze zmienionym identyfikatorem grupy.

3.6.5. Problem krzyżowych blokad komunikacji

Dopasowanie pakietu do generującego go procesu, czy to za pomocą `pid-owner`, czy też którejś z metod alternatywnych, identyfikuje jedynie nadawcę, nie mówi zaś nic o odbiorcy pakietu. Jak łatwo się przekonać, jest to informacja niewystarczająca do zaimplementowania dowolnych blokad komunikacji, gdy na pojedynczym węźle może być kilka procesów.

Załóżmy, że istnieją dwa komputery, oznaczone na rys. 3.1 *Pnode A* i *Pnode B* o adresach *AddrA* i *AddrB*, na każdym z nich uruchomione są dwa procesy, odpowiednio *Serwer 1*, *Serwer 2* oraz *Serwer 3*, *Serwer 4*. W takim systemie nie jest możliwe zablokowanie komunikacji między procesami *Serwer 1* i *Serwer 4*, jeśli równocześnie *Serwer 2* i *Serwer 3* mają się komunikować bez przeszkód.



Rysunek 3.1: Niezależne blokady komunikacji procesów na tym samym węźle

Chociaż *Pnode A* może zidentyfikować pakiety tworzone przez *Serwer 1*, to nie jest w stanie stwierdzić, czy ich odbiorcą będzie *Serwer 3* czy *Serwer 4* — w obu przypadkach adres docelowy to *AddrB*, natomiast numery portów przydzielone zostały dynamicznie.

Aby podjąć decyzję o porzuceniu pakietu, jeden z węzłów musi jednocześnie znać i nadawcę i odbiorcę pakietu. W kolejnych punktach opisane zostaną sposoby uzyskania informacji o obu stronach połączenia na jednym z węzłów.

3.6.6. Przesyłanie numerów portów

Jednym z rozwiązań jest poinformowanie wszystkich węzłów w systemie o portach przypisanych do każdego z procesów. Jest to rozszerzenie techniki poznawania przydzielonych procesowi numerów portów, opisanej w p. 3.6.4 powyżej jako alternatywa dla `pid-owner`. Poznane przez bibliotekę opakowującą numery portów są rozsyłane do wszystkich pozostałych

węzłów w systemie. Każdy z węzłów pamięta przydzielone procesom numery portów. Węzeł ma więc pełną informację potrzebną do zadecydowania o przepuszczeniu lub porzuceniu każdego pakietu — zna zarówno jego nadawcę (adres IP i port), jak i odbiorcę.

Zalety:

- Nie wymaga zmian w konfiguracji sieci ani nie wprowadza dodatkowych założeń na jej temat.

Wady:

- Rozgłaszanie do wszystkich węzłów w systemie informacji o każdym otwieranym porcie może wprowadzić duże opóźnienia na otwieranie połączeń sieciowych.
- Wymaga częstych zmian reguł iptables na wszystkich komputerach gdy zablokowany proces próbuje otwierać nowe połączenia.
- Sytuacja wyścigu pomiędzy otwieraniem połączenia a rozsyłaniem informacji o przydzielonym porcie może spowodować przepuszczenie kilku pakietów pomimo ustanowionej blokady.
- Trudno jest wykryć zamykanie połączeń.

3.6.7. Przesyłanie identyfikatora procesu

Na węźle wysyłającym pakiet można poznać tożsamość procesu nadawcy, np. korzystając z metody opisanej w punkcie 3.6.3. Tak więc zamiast budować bazę danych łączącą numery portów z procesem, można dołączyć unikatowy identyfikator procesu nadawcy do każdego wysyłanego przez ten proces pakietu, informując o tej tożsamości węzeł odbiorcy.

Można rozważyć wykorzystanie do tego celu któregoś z rzadko używanych pól nagłówka IP:

TTL

Jest to 8-bitowe pole oznaczające liczbę węzłów pośredniczących, przez jaką pakiet może jeszcze przejść, zanim zostanie porzucony. TTL inicjowane jest pewną wartością na węźle nadawcy i zmniejszane na każdym pośrednim węźle. Jeśli spadnie do 0 przed osiągnięciem celu, to pakiet jest porzucany.

Pole TTL służy do przeciwdziałania cyklom w trasie pakietu. W sieciach lokalnych, gdzie komputery są podłączone bezpośrednio lub przez niewielką liczbę węzłów pośrednich, można go użyć do zakodowania informacji.

Nie jest to jednak rozwiązanie wiarygodne. Wartość wysyłana jest inna niż odbierana, trzeba więc przewidzieć o ile może się zmniejszyć w drodze przez sieć. Co więcej, jeśli zmieni się konfiguracja sieci, wartości TTL przestaną być poprawnie odczytywane po stronie odbiorcy.

TOS

TOS jest także 8-bitowym polem nagłówka IP. Według standardu miało ono służyć do wyboru typu obsługi pakietu przez sieć — przykładowo, czy maksymalizowana ma być przepustowość, czy minimalizowane opóźnienie bądź koszt.

Wartości tego pola są obecnie ignorowane przez większość urządzeń sieciowych. Można je za to bez problemu ustawiać w interfejsie gniazd (funkcją `setsockopt()` z parametrem `IP_TOS`). Istnieje także moduł do iptables, dopasowujący reguły do pakietów z konkretną

wartością pola TOS. Zachęcająco zatem zdaje się wyglądać użycie tego pola do przesyłania dodatkowych danych o pakiecie.

Okazuje się jednak, że takie podejście, jeśli nawet działa z obecnymi urządzeniami i systemami operacyjnymi, może mieć wpływ na sposób przetwarzania pakietów w przyszłości. Kilka bitów pola TOS zostało niedawno przedefiniowanych i jest używanych do ECN (*explicit congestion notification*) — mechanizmu kontroli przepływu, który informuje o przepełnieniu sieci bez konieczności porzucania pakietów. Prawdopodobnie także reszta pola TOS otrzyma nowe przeznaczenie albo zacznie być wykorzystywana zgodnie z oryginalnym.

Opcje protokołu IP

Opcje to dodatkowe informacje, które mogą być przesyłane w specjalnej części nagłówka IP o zmiennej długości. Każda opcja składa się z 5-bitowego numeru opcji oraz związanych z nią danych. Długość danych jednej opcji może być zmienna, jednak sumaryczna reprezentacja wszystkich opcji w pakiecie nie może przekraczać 40 bajtów.

Interfejs gniazd umożliwia zdefiniowanie opcji IP wysyłanych w pakietach za pomocą funkcji `setsockopt()`. Możliwe jest więc dodawanie do pakietów generowanych przez testowany proces opcji identyfikującej nadawcę, np. wywołując `setsockopt()` w przechwyconych wywołaniach funkcji `socket()`.

Narzut związany z przesyłaniem identyfikatora w opcji IP jest bardzo niewielki — wynosi zaledwie 2 bajty na pakiet (plus rozmiar samego identyfikatora).

Rozwiązanie to ma jednak szereg wad.

Spośród 32 możliwych identyfikatorów opcji 25 zostało oficjalnie zarejestrowanych przez organizację IANA. Pozostałe 7 numerów opcji nie zostało jeszcze przydzielonych, jednak ich użycie w bibliotece testującej może spowodować konflikt z zastosowaniami, które zostaną dopiero zarejestrowane w przyszłości.

Ponadto, wbrew zaleceniom standardu IP [7], część urządzeń sieciowych niepoprawnie reaguje na pakiety zawierające niezrozumiałe opcje. Zamiast przesłać je dalej w niezmienionej formie, urządzenia takie usuwają opcje z pakietu, a czasem nawet porzucają cały pakiet.

Kolejnym problemem są braki w module dopasowującym iptables. Jako kryterium dopasowania pakietu do reguły można zdefiniować istnienie w nim opcji o określonym numerze, ale nie ma możliwości sprawdzenia danych związanych z opcją. Oznacza to, że każdy proces musiałby być identyfikowany oddzielnym numerem opcji, których pozostało zaledwie 7.

Kapsułkowanie

Wykorzystywanie istniejących pól nagłówków IP nie jest rozwiązaniem akceptowalnym, gdyż ich modyfikacja może spowodować niepoprawne działanie sieci.

Aby dodać dodatkowe informacje bez ingerencji w sam pakiet, można się posłużyć *kapsułkowaniem*. Kapsułkowanie polega na przesyłaniu kompletnych pakietów IP wewnątrz innego połączenia sieciowego — pakiety są wówczas traktowane jak zwykłe dane.

Zasada działania tego mechanizmu jest prosta. Nadawca przechwytuje pakiet, który ma zostać wysłany w sieć, wysyła go po innym połączeniu sieciowym do odbiorcy, który następnie odpakowuje pakiet i symuluje jego odebranie przez interfejs sieciowy. Droga, jaką faktycznie przebył pakiet w sieci jest dla niego niezauważalna. Kapsułkowanie często wykorzystuje się do połączenia dwóch sieci, które nie mogą się ze sobą komunikować bezpośrednio. Stąd też inną nazwą tego mechanizmu jest *tunelowanie*.

Z każdym mechanizmem tunelowania związany jest drobny narzut. Pakiety opakowane są większe, mają bowiem dwie kopie nagłówków IP — zewnętrzną, widoczną podczas drogi przez

sieć, oraz wewnętrzną, ukrytą w części z danymi. Standardowy rozmiar nagłówka IP to 20 bajtów, do tego dochodzą informacje konieczne do rozróżnienia różnych tuneli (w przypadku opisanych dalej tuneli GRE są to zwykle 4 bajty). Standardowo maksymalny rozmiar pakietu w sieciach *Ethernet* to 1500 bajtów. Przy korzystaniu z tunelowania przepustowość sieci spadnie więc o $(20 + 4)/1500 = 1.6\%$.

Tunele GRE

W standardowym jądrze Linuksa istnieje kilka modułów służących do tworzenia tuneli. Najpopularniejsze są tunele GRE. Są one zgodne ze standardem używanym przez różne urządzenia sieciowe, dlatego ich użycie jest zalecane.

Każdy tunel tworzony jest poprzez zdefiniowanie adresu węzła będącego drugim końcem tunelu. W efekcie do systemu dodawany jest nowy interfejs sieciowy, a wszystkie pakiety wysyłane tym interfejsem są opakowywane w pakiety IP specjalnego typu i wysyłane do odbiorcy.

W tradycyjnych tunelach przesyłane są same pakiety, bez żadnych dodatkowych danych. Z punktu widzenia przesyłania tożsamości nadawcy, stosowanie tuneli GRE nie daje wiele. Tunele takie mogą natomiast być przydatne do uproszczenia konfiguracji sieci w innych technikach blokowania komunikacji, opisanych w dalszej części pracy.

3.6.8. Kapsułkowanie w trybie użytkownika

Dzięki interfejsom QUEUE oraz TUN/TAP, tunel można zaimplementować w trybie użytkownika. Mając pełną kontrolę nad przesyłanymi danymi, można do nich dołączyć identyfikator procesu nadawcy.

Najbardziej naturalnym mechanizmem przechwytywania pakietów wygenerowanych przez testowane procesy zdaje się być TUN. Niestety, po wysłaniu pakietu do trybu użytkownika, tracony jest cały kontekst pakietu i możliwość przypisania go do nadawcy.

Korzystając z biblioteki `libipq`, proces dostaje dodatkowo metadane pakietu. Szczególnie ciekawym polem jest `fwmark` — jest to liczbowy znacznik związany z każdym pakietem w jądrze Linuksa. `Fwmark` może być ustawiany w regułach iptables, może być też użyty jako klucz przy dopasowywaniu pakietów. W bibliotece blokującej komunikację, wartość `fwmark` może oznaczać identyfikator procesu, rozpoznany jedną z metod z punktu 3.6.3.

Niestety, pakietów dostarczonych przez `libipq` nie można po prostu przesłać po sieci i zaakceptować na zdalnym węźle. Nie miałoby to zresztą większego sensu — wywołanie celu QUEUE występuje w konkretnym miejscu w łańcuchu reguł iptables, które na innym węźle mogą wyglądać zupełnie inaczej.

Okazuje się jednak, że można połączyć oba mechanizmy: `libipq` może służyć do przechwytywania i identyfikowania wysyłanych pakietów, natomiast TUN do „wstrzykiwania" ich po stronie odbiorcy. Reguły do przechwytywania pakietów mogą wyglądać następująco:

```
iptables -A OUTPUT -m owner --uid-owner uid_procesu \
    --gid-owner gid_procesu -j MARK --set-mark unikatowy_id_procesu

iptables -A OUTPUT -m owner --uid-owner uid_procesu \
    --gid-owner gid_procesu -j QUEUE
```

Implementacja logiki blokowania komunikacji oraz przesyłanie opakowanych pakietów do odbiorcy może odbywać się w procesie odbierającym zakolejkowane komunikaty. Do komunikacji między procesami tunelującymi na różnych węzłach najlepiej jest użyć połączenia UDP,

ale równie dobrze (choć z nieco większym narzutem) zadziała TCP. Po wysłaniu pakietu wraz z identyfikatorem nadawcy do procesu odpakowującego na węźle odbiorcy, zakolejkowany pakiet powinien być porzucony, aby nie doszedł w dwóch kopiach — raz rozpakowany z tunelu, a drugi raz standardowo, po sieci.

Proces odbierający po otrzymaniu pakietu i identyfikatora nadawcy, może zapisać pakiet do urządzenia TUN, z którego rozpocznie się jego normalna obsługa w jądrze.

W powyższym scenariuszu jest niestety poważna luka — proces odbierający dane z tunelu zna co prawda nadawcę pakietu, ale nie bardzo wiadomo jak miałby poznać jego lokalnego odbiorcę. Standardowe metody oparte na module `owner-match` (por. p. 3.6.3) działają wyłącznie dla pakietów wychodzących. Z kolei zapamiętywanie otwartych portów nie jest w pełni wiarygodne.

W najnowszej wersji iptables dołączona jest łata rozszerzająca działanie `owner-match` tak, żeby możliwe było dopasowanie po właścicielu gniazda odbierającego pakiet. Niestety, funkcjonalność ta nie została jeszcze dodana do standardowej dystrybucji Linuksa i wymaga zaaplikowania łaty. Metody przesyłające tożsamość nadawcy na zdalny węzeł staną się rzeczywiście użyteczne dopiero wtedy, gdy będzie można skorzystać z tego rozszerzenia lub jeśli uda się rozwiązać problem przypisania pakietu do odbiorcy w inny sposób.

3.6.9. Przypisanie procesom puli portów

Zamiast rozsyłać informacje o tym, które porty zostały przydzielone do którego procesu, można z góry przydzielić każdemu procesowi pulę portów, z których może korzystać. Biblioteka opakowująca może przechwytywać wywołania otwierające połączenia i związać tworzone gniazda z portami przydzielonym z tej właśnie puli. Dzięki temu całkowicie pominięty zostanie mechanizm losowego przydział portów przez jądro.

Jeśli dobrane zostaną rozłączne zakresy portów, to każdy testowany proces będzie rozpoznawalny u odbiorcy komunikatu po źródłowym adresie IP i numerze portu.

Pozostaje kwestia gwarancji, że żaden inny proces nie użyje, na przykład przez losowy przydział, portu z puli zarezerwowanej dla blokowanego procesu. Niestety, w standardowym jądrze Linuksa nie ma możliwości zarezerwowania portów dla konkretnego procesu — jeśli numer portu podany w wywołaniu `bind()` nie jest zajęty, to każdy proces może z niego skorzystać (wyjątkiem od tej reguły są porty o numerach niższych niż 1024 — korzystać z nich mogą tylko procesy systemowe).

Aby częściowo rozwiązać ten problem, można ustalić pulę portów przypisanych procesowi poza zakresem przydzielanym automatycznie przez jądro. Zakres ten ustawiany jest za pomocą interfejsu `sysctl`, a dokładnie w pliku `/proc/sys/net/ipv4/ip_local_port_range`. Czytając ten plik można sprawdzić granice obecnie używanego zakresu.

Oczywiście powyższe rozwiązanie nie zabezpiecza przed procesami, które same wybierają używany port i nie korzystają z przydziału automatycznego. Jeśli inny proces wykorzysta port z puli przydzielonej testowanemu procesowi, to ewentualna blokada komunikacji będzie dotyczyła także tego procesu.

Szczęśliwie, większość procesów używa niewielkiej liczby portów o ustalonych numerach. Są to w zdecydowanej większości porty nasłuchujące, których numery muszą być ustalone, aby umożliwić odnalezienie się procesów w sieci. W większości istniejących aplikacji sieciowych portów nasłuchujących jest niewiele, wobec tego tylko drobna liczba gniazd będzie tworzona z pominięciem dynamicznego przydziału jądra. Prawdopodobieństwo kolizji z przypisanym do testów zakresem portów, a szczególnie zużycia całego zakresu przez procesy niezwiązane z testem jest więc znikome.

Co więcej, procesy nie uczestniczące w teście nie powinny w ogóle komunikować się z procesami testowanymi — gdyby taka komunikacja następowała, nie byłaby ona pod kontrolą scenariusza testu. Procesy aktywnie poszukujące otwartych portów są niebezpieczne nawet pomijając problematykę blokowania komunikacji, ponieważ mogą zaburzyć wynik testu, wysyłając do testowanej aplikacji niespodziewane dane.

Wobec powyższych faktów, problem zarezerwowania portów do wyłącznego użycia przez testowany proces nie jest aż tak istotny — wystarczy, aby procesy biorące udział w teście miały przydzielane numery portów z rozłącznych zakresów.

Wady:

- Z góry ograniczona jest maksymalna liczba równocześnie używanych przez aplikację portów.
- Konieczne jest zdefiniowanie puli dostępnych portów, uwzględniając uruchomione serwisy sieciowe oraz współistnienie testów różnych użytkowników.
- Blokowanie komunikacji biorące pod uwagę wyłącznie użyte zakresy portów mogłoby błędnie przerwać łączność między procesami nie należącymi do testu, a komunikującymi się po portach należących do blokowanego zakresu. Konieczne jest więc dodatkowe sprawdzenie, czy lokalny proces należy do środowiska testu.

Zalety:

- Wystarczy jednokrotnie rozesłać informacje o przydzielonych zakresach portów na początku testu. Wpływ na wydajność jest pomijalny — i tak na początku testu trzeba skontaktować się ze wszystkimi węzłami, aby uruchomić procesy.
- Rozwiązanie nie wymaga rekonfiguracji sieci i jest niezależne od jej topologii.

3.6.10. Przypisanie procesom różnych adresów źródłowych

Innym sposobem rozróżniania pakietów należących do poszczególnych procesów może być przypisanie procesom różnych źródłowych adresów IP. Mechanizm ten jest nieco odmienny od pozostałych, bowiem nie są blokowane pojedyncze procesy, ale całe interfejsy sieciowe. Podejście to jest bardziej realistyczne i wygląda spójnie dla pozostałych uczestników testu — blokada każdego procesu odpowiada tu uszkodzeniu pojedynczego interfejsu sieciowego.

Przypisanie dodatkowych adresów IP do węzła

W standardowej konfiguracji, każdy z komputerów testujących ma tylko jeden interfejs sieciowy, z przypisanym do niego pojedynczym adresem IP. Pierwszym krokiem jest więc zmuszenie systemu operacyjnego do używania kolejnych, sztucznych adresów.

W Linuksie istnieje kilka sposobów utworzenia dodatkowych interfejsów sieciowych. Kilka z nich to:

- Skorzystanie z modułu `dummy`. Jest to „pusty” sterownik interfejsu sieciowego, który ignoruje wszystkie wysyłane komunikaty i nigdy nie odbiera żadnych z sieci. Pakiety skierowane do adresu IP przypisanego temu interfejsowi zostaną oczywiście odebrane poprawnie, ponieważ pakiety dla lokalnych adresów nie przechodzą przez sterownik.
- Przypisanie wielu adresów IP do jednego, istniejącego interfejsu. W dawnych wersjach Linuksa mechanizm ten nazywany był tworzeniem *aliasów* interfejsu podstawowego. Interfejs z przypisanymi wieloma adresami IP traktuje je równoważnie — akceptuje pakiety przychodzące i generuje wychodzące z dowolnym z pasujących adresów.

- Tunelowanie. Ten mechanizm tworzy wirtualny interfejs, który przesyła pakiety do zdalnego węzła korzystając z interfejsów rzeczywistych. Dokładniejszy opis tunelowania znajduje się w p. 3.6.7.

Różnice między tymi mechanizmami zostaną opisane w dalszej części tego rozdziału.

Osobną kwestią jest wybór adresu. W Internecie zdefiniowanych jest kilka zakresów adresów „prywatnych", które nie zostaną nigdy przydzielone i mogą służyć do tworzenia prywatnych sieci. W wielu przypadkach sieci lokalne już używają któregoś z tych zakresów, należy więc tak dobrać wykorzystane adresy IP, żeby nie spowodować konfliktów z komputerami dostępnymi w sieci lokalnej.

Mechanizm wyboru adresu źródłowego na węźle z wieloma interfejsami sieciowymi

Na węźle z wieloma adresami IP, któryś z adresów musi zostać wybrany jako adres źródłowy. Adres ten wpisywany jest do nagłówków IP wysyłanych pakietów. Wybór adresu źródłowego dokonywany jest niezależnie dla każdego gniazda IP. Przy podejmowaniu decyzji brane są pod uwagę dwie reguły:

- Gniazda, które zostały związane z konkretnym adresem IP poprzez wywołanie `bind()` korzystają z adresu podanego w tym wywołaniu. Oczywiście, w kodzie funkcji systemowej `bind()` sprawdzana jest poprawność adresu i to, czy należy on do zbioru adresów IP tego węzła[1].

- Dla gniazd, które nie były związane z żadnym konkretnym adresem IP, w momencie otwierania połączenia wybierany jest adres będący „najbliżej" zdalnego węzła. Wybór ten dokonywany jest na podstawie tablic trasowania.

W obu przypadkach, w momencie nawiązania połączenia adres źródłowy gniazda jest już ustalony i nie zmienia się przez cały czas trwania tego połączenia. Wszystkie pakiety generowane przez gniazdo otrzymują jego adres źródłowy. Z kolei pakiety przychodzące przekazane będą do gniazda wyłącznie jeśli jego adres źródłowy zgadza się z adresem docelowy pakietu (dodatkowo, w protokołach TCP i UDP zgadzać się muszą numery portów).

Szczególnym przypadkiem są gniazda nasłuchujące, nie związane z żadnym konkretnym adresem IP. Akceptują one pakiety z dowolnym z lokalnych adresów IP, o ile tylko zgadza się numer portu i nie ma żadnego lepiej pasującego gniazda (np. z już ustanowionym połączeniem).

Z powyższego opisu wynika kilka właściwości procesów uruchomionych na maszynach z wieloma adresami IP:

- Gniazda pojedynczego procesu mogą mieć wiele różnych adresów źródłowych.

- Jeśli proces nie wykonał `bind()` na gnieździe podczas aktywnego otwierania połączenia, to jako adres źródłowy zostanie wybrany adres najbliższy adresowi docelowemu.

- Jeśli proces poprawnie wykonał `bind()` na gnieździe, to adresem źródłowym wszystkich pakietów wychodzących z tego gniazda będzie adres podany w wywołaniu `bind()`.

- Gniazdo, które zostało utworzone w wyniku akceptacji połączenia przez proces nasłuchujący na macierzystym gnieździe o adresie `INADDR_ANY`, zostanie związane z adresem najbliższym adresowi węzła inicjującego połączenie.

[1] W rzeczywistości możliwe jest związanie gniazda z adresem, który nie należy do węzła. Wymaga to jednak ustawienia flagi `/proc/sys/net/ipv4/ip_nonlocal_bind`, która ze względu na zgodność z istniejącymi programami sieciowymi, jest domyślnie wyłączona.

Przypisanie procesu do konkretnego adresu IP

Aby zmusić testowany proces do korzystania z konkretnego adresu IP, trzeba z powyższego procesu wyeliminować wszystkie przypadki, w których wybór adresu dokonywany jest automatycznie, na podstawie tablicy trasowania.

Dokonać tego można przechwytując:

- Wywołania wszystkich funkcji systemowych otwierających połączenie, czyli:
  - `connect()`,
  - `sendto()`,
  - `sendmsg()`.

  Każda z tych funkcji, jeśli przekazane do niej gniazdo nie było dotąd związane, dokonuje automatycznego wyboru adresu źródłowego. Aby do tego nie dopuścić, wystarczy przed pierwszym wywołaniem tych funkcji związać gniazdo z adresem IP przypisanym procesowi.
- Wywołania funkcji `bind()`, aby:
  - zgłosić błąd przy próbie przypisania gniazdu innego adresu,
  - ustawić także adres jeśli wywołanie `bind()` ustawiało jedynie numer portu.
- Wywołania funkcji `listen()`, aby przy braku poprzedzającego `bind()` przypisać do nasłuchu jedynie dozwolony adres.

Do przechwycenia wywołań najlepiej nadaje się biblioteka dynamiczna przykrywająca odpowiednie symbole podczas ładowania.

Procesy przypisane do konkretnego adresu IP nie mogą utworzyć gniazda do nasłuchiwania jednocześnie na wszystkich interfejsach sieciowych. Wywołania funkcji `listen()` z parametrem `INADDR_ANY` zostaną podmienione tak, aby połączenia przyjmowane były jedynie na zdefiniowanym adresie IP. Takie zachowanie jest niezbędne, bowiem w przeciwnym razie zaakceptowane połączenia związane byłyby z adresem najbliższym drugiej stronie połączenia, bez możliwości zmiany tego wyboru. Powyższe zachowanie `listen()` stwarza pewien problem — proces nie będzie odpowiadał na próby połączeń przez interfejs *loopback* (na adresie lokalnym `127.0.0.1`). Niestety w Linuksie proces nasłuchujący na pojedynczym gnieździe nie może akceptować połączeń z kilkoma konkretnymi adresami.

Zmiana nazwy węzła

Poza modyfikacją funkcji zarządzających gniazdami, konieczna może być także zmiana nazwy węzła. Do pobierana tej nazwy służy funkcja systemowa `gethostname()`. Często nazwa węzła, zamieniona na adres IP, traktowana jest przez programy sieciowe jako „główny" adres, identyfikujący lokalny węzeł. W rzeczywistości jest to kwestia umowna — w jądrze żaden z interfejsów nie jest szczególnie uprzywilejowany. Mimo to wiele aplikacji, włącznie z przedstawioną w tej pracy, właśnie na adresie opartym na `gethostname()` przyjmuje połączenia i rozsyła go w komunikatach jako adres zwrotny.

Ponieważ nazwa węzła jest taka sama dla wszystkich procesów uruchomionych na komputerze, konieczne jest przechwycenie wywołania `gethostname()`. Zamiast rzeczywistej nazwy można przekazać napisową reprezentację adresu IP przypisanego do procesu — dzięki temu nie trzeba modyfikować konfiguracji serwera nazw.

Zapewnienie łączności z dodatkowymi adresami

Po utworzeniu nowego adresu IP i przypisaniu do niego procesu okazuje się, że proces ten stracił kontakt z wszystkimi węzłami poza lokalnym. W rzeczywistości to pozostałe węzły straciły możliwość wysyłania pakietów do testowanego procesu. Przyczyną problemu jest brak odpowiedniej trasy w tablicach tras zdalnych węzłów. W efekcie albo w ogóle nie istnieje ścieżka do nowego adresu, albo pakiety przesyłane są do domyślnej bramki łączącej sieć lokalną z resztą świata.

Aby zapewnić łączność z nowymi adresami IP trzeba więc zmodyfikować tablice trasowania. Możliwe rozwiązania zależą od wybranego sposobu dodania adresu:

- Jeśli nowe adresy stworzone zostały za pomocą aliasów oraz jeśli użyte prywatne adresy IP tworzą odrębną sieć IP, to wystarczy dodać trasę do tej sieci przez interfejs, na którym zdefiniowane zostały aliasy:

```
route add -net $WSPOLNY_PREFIKS_SZTUCZNYCH_ADRESOW_IP \
    netmask $MASKA_SZTUCZNEJ_SIECI dev eth0
```

- Jeśli nowe adresy przypisane są do interfejsów `dummy` lub zostały gorzej dobrane, to trzeba stworzyć ścieżkę do każdego ze sztucznych adresów. Jako bramkę trzeba podać rzeczywisty adres IP węzła, na którym dodano sztuczny adres:

```
route add -host $SZTUCZNY_IP1_NA_WEZLE1 gw $PUBLICZNY_IP_WEZLA1
route add -host $SZTUCZNY_IP2_NA_WEZLE1 gw $PUBLICZNY_IP_WEZLA1
...
route add -host $SZTUCZNY_IP1_NA_WEZLE2 gw $PUBLICZNY_IP_WEZLA2
...
```

- Gdy węzły nie są bezpośrednio połączone w sieci lokalnej, można stworzyć tunele między każdą parą węzłów. Na każdym węźle będzie aż $liczba\_sztucznych\_adresów\_na\_tym\_węźle * (liczba\_węzłów - 1)$ tuneli.

Izolacja procesów spoza testu

W normalnym jądrze nie da się zabronić niezwiązanemu procesowi korzystania z testowego interfejsu.

Nie jest to jednak duży problem z punktu widzenia izolacji procesów spoza testu. Jeśli proces nie komunikuje się z testowanymi serwerami, to nie powinien w ogóle używać nowych adresów.

Jeśli jest to konieczne, to można użyć reguł iptables, które dla wszystkich procesów próbujących wysyłać po testowym interfejsie sieciowym, a nie należących do testu, dokonają translacji adresu źródłowego (SNAT) na adres publiczny węzła. Do wykrycia czy proces należy do testu czy nie można użyć techniki opisanej w p. 3.6.3.

Wady i zalety

Zalety:

- obraz systemu podczas blokad komunikacji jest spójny — blokada dotyczy wszystkich procesów na interfejsie,
- można blokować pakiety zarówno u nadawcy, jak i odbiorcy,
- można uruchomić na jednym węźle procesy mające na stałe przypisany numer portu nasłuchującego.

Wady:

- konieczne jest zdefiniowanie wolnych adresów przez administratora,
- trzeba zsynchronizować adresy używane przez poszczególne węzły,
- wersja w pełni efektywna (bez tunelowania) działa tylko w sieci lokalnej, w której nie ma bramek,
- przy tunelowaniu konieczne jest utworzenie tunelu między każdą parą węzłów,
- z procesem przypisanym do konkretnego adresu nie można się komunikować przez lokalny adres *loopback* 127.0.0.1.

3.6.11. Podsumowanie

Podsumujmy funkcje udostępniane przez poszczególne mechanizmy blokowania komunikacji w niezmienionym jądrze:

1. `pid-owner` na maszynach jednoprocesorowych lub `uid-owner` z `gid-owner` na wieloprocesorowych
   + Pakiet pasuje do tych reguł wtedy i tylko wtedy, gdy co najmniej jedna ze stron komunikacji (a dokładniej nadawca) należy do testowanego systemu.
   – Nie działają po stronie odbiorcy, a więc nie dają możliwości wydajnego blokowania.
2. Rozszerzona wersja `uid-owner` z `gid-owner` z nowej wersji iptables, która działa także w łańcuchu INPUT:
   + Pakiet pasuje do tych reguł wtedy i tylko wtedy, gdy co najmniej jedna ze stron komunikacji należy do testowanego systemu.
   + Działa zarówno po stronie odbiorcy, jak i nadawcy, choć dotyczy tylko procesów na lokalnym węźle.
3. Pula portów:
   + Rozróżnia pomiędzy różnymi procesami należącymi do testu (o ile tylko pule są rozłączne w ramach jednego węzła).
   + Działa zarówno na węźle nadawcy jak i odbiorcy.
   – Nie daje pewności, że pasujący pakiet faktycznie należy do testu, a nie jest komunikacją postronnych procesów.

4. Kapsułkowanie w trybie użytkownika poprzez libipq:

   Wymaga metody rozpoznania tożsamości lokalnego procesu nadającego pakiet.

   \+ Przesłanie (dowolnej) informacji o tożsamości procesu do węzła docelowego.

5. Przypisanie procesom prywatnych adresów IP:

   \+ Rozróżnia pomiędzy różnymi procesami należącymi do testu.

   \+ Działa zarówno u odbiorcy, jak i nadawcy.

   \+ Izolacja od procesów postronnych (przy dodatkowych regułach SNAT).

3.7. Symulacja zniszczenia węzła

Symulację zniszczenia węzła można uzyskać w prosty sposób, wprowadzając blokadę komunikacji trwającą aż do końca testu. Aby zaoszczędzić zasoby komputera testującego, odłączone procesy mogą zostać następnie zakończone — nie wpłynie to na resztę systemu.

3.8. Symulacja awarii zasilania

Z punktu widzenia testowanego procesu, awaria zasilania ma wpływ na dwa elementy systemu:

- Powoduje natychmiastowe zaprzestanie wykonywania procesu, bez wykonania żadnego kodu zapamiętującego stan.
- Powoduje utratę zawartości buforów dyskowych systemu operacyjnego. Dane jeszcze nie zapisane na bezpiecznym nośniku zostają utracone.

Zatrzymanie procesu jest efektem oczywistym i prostym do zasymulowania — wystarczy zakończyć testowany proces wysyłając do niego nieprzechwytywalny sygnał `SIGKILL`.

Natomiast utrata nie zapisanych na dysku buforów jest bardzo istotnym czynnikiem wpływającym na bezpieczeństwo danych przechowywanych przez system. Aby zapewnić poprawne funkcjonowanie po awarii zasilania, testowany system stosuje dziennik zmian. Wszelkie zmiany danych dyskowych, zanim zostaną zlecone do zapisania na dysku, są najpierw zapisywane do dziennika. W przypadku awarii proces może je odtworzyć czytając dziennik.

Samo zakończenie procesu nie powoduje utraty buforów, gdyż zajmuje się nimi system operacyjny. W dalszej części punktu zostaną więc opisane sposoby symulacji także tego elementu.

3.8.1. W maszynie wirtualnej

Olbrzymią zaletą maszyn wirtualnych jest dość wierna i prosta symulacja awarii zasilania — na ogół wystarczy zakończyć proces maszyny wirtualnej, najlepiej nieprzechwytywalnym sygnałem `SIGKILL`. Zatrzymany w ten sposób zostanie cały system operacyjny, wraz z testowanymi procesami. Wszelkie dane zbuforowane w jądrze wewnętrznego systemu zostaną utracone.

Niestety, ze względu na problemy opisane w p. 3.4, uruchomienie testowanych procesów pod maszyną wirtualną nie jest sensowne.

3.8.2. Restart komputera testującego

Dość prostym sposobem wprowadzenia tej usterki jest wymuszenie restartu komputera testującego. W Linuksie można w tym celu posłużyć się programem `reboot`, używanym na ogół do tzw. „czystego" restartu komputera.

Okazuje się, że w `reboot` może wykonać także natychmiastowy restart, bez zapisywania buforów. Służą do tego opcje `-f` — bezpośredni restart bez zatrzymywania uruchomionych serwisów — oraz `-n` — nie zapisuj buforów przed restartem.

Ograniczenia

Oczywiście operacja ta dotyczy całego komputera, nie jest więc możliwe jej wykonanie tylko dla wybranego procesu bądź urządzeń blokowych.

Co więcej, restart komputera nie uwzględnia wpływu buforowania przeprowadzanego przez urządzenia dyskowe. Niestety, z problemem tym niewiele da się zrobić — nie ma programowej metody wyłączenia zasilania dysku. Zresztą częste cykle włączania i wyłączania znacznie przyspieszyłyby awarię dysku używanego w teście.

3.8.3. Porzucanie zapisów przez urządzenie blokowe

W jądrach wersji 2.2 i 2.4 istniały łaty na Linuksa służące do testowania systemów plików korzystających z dzienników[9, 15]. Rozszerzały one standardowe urządzenie blokowe `loop`, dodając do niego porzucanie zlecanych zapisów i zgłaszanie błędów przy odczycie.

Pewnym rozwiązaniem byłoby więc zaimplementowanie podobnej funkcjonalności w jądrze 2.6 (użycie Linuksa 2.6 jest wymagane przez testowaną aplikację).

Aby zasymulować awarię zasilania, trzeba by połączyć blokadę wychodzącej komunikacji z powyższym mechanizmem porzucania zapisów.

3.8.4. Podsumowanie symulacji awarii zasilania

Symulacja awarii zasilania zawsze powoduje błędy niedeterministyczne i trudne w odtworzeniu poprzez ponowne wykonanie tego samego scenariusza. Z drugiej strony, ponieważ po zakończeniu procesu cały jego stan zaszyty jest w zapisanych na dysku danych, można prosto zrobić jego kopię dla celów późniejszej analizy. Dlatego, w odróżnieniu od blokady komunikacji, przy awarii zasilania mniej istotne jest uruchamianie kilku testowanych procesów na jednej maszynie — programiści nie muszą powtarzać zakończonych błędem scenariuszy w czasie dnia, nie ma więc ryzyka interakcji między równoczesnymi testami.

Testowana aplikacja wykorzystuje bezpośrednie odwołania do dysku. Procesy korzystające z flagi O_DIRECT pomijają pamięć buforową systemu operacyjnego przy odwołaniach do urządzeń. Dla takich procesów wystarczającą symulacją jest nagłe zakończenie procesu poprzez wysłanie do niego sygnału `SIGKILL`.

Mechanizm porzucania zapisów, choć działa mniej inwazyjnie niż `reboot`, wymaga pracochłonnego utrzymywania łaty na jądro Linuksa, a daje tylko niewielki zysk funkcjonalności. Tak więc to `reboot` można uznać za najlepszy mechanizm symulowania awarii zasilania węzła.

3.9. Podsumowanie

Najlepsze z technik opisanych w tym rozdziale posłużyły do implementacji biblioteki symulującej awarie, która jest częścią tej pracy.

W wyniku analizy wydajności, wierności i stopnia skomplikowania różnych metod wprowadzania awarii, najlepszymi okazały się metody modyfikujące zachowanie rzeczywistego jądra Linuksa. Co więcej, zmiany te nie wymagają ingerencji w kod jądra, gdyż wykorzystują tylko interfejsy już udostępniane przez różne części Linuksa.

Rozdział 4

Środowisko STAF + STAX

W tym rozdziale opisuję środowisko STAF oraz jego rozszerzenie STAX. Biblioteka do wprowadzania awarii, która powstała w wyniku tej pracy, działa właśnie pod kontrolą środowiska STAX. Wybór STAX podyktowany był głównie jego użyciem w już istniejących testach naszego systemu — dzięki temu udało się łatwo rozszerzyć te testy tak, aby korzystały z funkcji wprowadzania usterek. Niezależnie od tego, STAF wraz ze STAX okazał się bardzo wygodnym narzędziem do testowania systemów rozproszonych.

4.1. STAF

STAF (*Software Testing Automation Framework*) [12, 20] jest to system do automatyzacji zadań, a szczególnie testów. Główne cele, jakie przed sobą postawili twórcy STAF, to rozszerzalność i możliwość wielokrotnego wykorzystania komponentów systemu.

Duży nacisk położono na przenośność środowiska — STAF działa pod wieloma systemami operacyjnymi, m.in. Linuksem, Windows oraz Solaris. Zawiera także wbudowane wsparcie dla *unicode* i wielu stron kodowych, co jest istotne przy testowaniu różnych wersji językowych systemów.

STAF powstał w IBM jako system do automatyzacji testów OS/2. Obecnie rozpowszechniany jest na licencji OpenSource. Jest to środowisko dość dojrzałe — poza testami w IBM, wykorzystywany jest między innymi w *Linux Test Project* [17], czyli testach jądra Linuksa.

4.1.1. Architektura

Środowisko STAF jest oparte na serwisach. Każdy serwis udostępnia zestaw powiązanych ze sobą funkcji, takich jak zapisywanie komunikatów do dziennika (*log*), uruchamianie procesów, zarządzanie konfiguracją.

Cała funkcjonalność środowiska zaimplementowana została w serwisach. Szkielet STAF jedynie rozdziela żądania do odpowiednich serwisów oraz zapewnia komunikację — lokalną (z wykorzystaniem łączy nazwanych FIFO) oraz zdalną (korzystającą z TCP).

Począwszy od wersji 3.0, zmodularyzowany został także interfejs komunikacji.

Głównym elementem STAF jest tzw. *demon* STAFProc. Jest to proces, który działa w tle i wykonuje odebrane od klientów żądania. Poza STAFProc w skład pakietu wchodzą biblioteki umożliwiające komunikację ze STAF dla różnych języków programowania. Jednym z takich interfejsów jest samodzielny program STAF, który łączy się z serwerem na podanej maszynie i wykonuje żądanie wczytane z wiersza poleceń.

4.1.2. Komunikacja

Wszystkie funkcje serwisów STAF wykonywane są na zasadzie żądanie-odpowiedź. Klient, za pomocą prostej biblioteki (specyficznej dla języka programowania), wysyła tekstowe polecenie do podanego przez siebie serwisu, a w wyniku dostaje odpowiedź (także tekstową).

Cała komunikacja klientów ze STAF oraz jego serwisami odbywa się wyłącznie przez tekstowe polecenia. Była to świadoma decyzja projektowa autorów — dzięki temu możliwe jest korzystanie ze STAF z programów napisanych w wielu różnych językach programowania oraz ze skryptów. Gdyby udostępniany interfejs był binarny i specyficzny dla każdego serwisu, przenoszenie nowszych wersji biblioteki na wszystkie wspierane języki zajmowałoby dużo czasu.

Aby ułatwić przekazywanie bardziej skomplikowanych informacji, w wersji 3.0 wprowadzono mechanizm reprezentacji struktur danych w formie tekstowej. Pozwala to usprawnić przekazywanie np. słowników i list, jednocześnie zachowując tekstowy charakter interfejsu.

STAF udostępnia mechanizm komunikacji między różnymi maszynami, na których uruchomiony został STAFProc — na dowolnej maszynie można skorzystać z serwisu udostępnianego przez inną.

Aby poprawić bezpieczeństwo takiego rozwiązania, wprowadzony został prosty mechanizm autoryzacji. Każdemu zdalnemu komputerowi przypisany jest poziom bezpieczeństwa (*Trust Level*), wyrażony liczbą całkowitą od 0 do 5. Wymagany poziom bezpieczeństwa zdefiniowany jest dla każdej operacji udostępnianej przez serwisy. Tak więc, komputer, któremu przypisano poziom 0 nie ma żadnego dostępu, natomiast ten o poziomie 5 może wykonywać wszystkie operacje.

Postać żądań

Każde żądanie dla serwisów STAF składa się z:

- nazwy maszyny, na której żądanie ma zostać wykonane (lub `local` — na lokalnej maszynie),
- nazwy serwisu,
- nazwy polecenia (specyficznej dla serwisu) oraz
- opcjonalnej listy dodatkowych parametrów żądania.

Najprostszym sposobem wykonywania żądań jest skorzystanie z programu STAF:

```
STAF nazwa_maszyny nazwa_serwisu polecenie_dla_serwisu dodatkowe_parametry...
```

4.1.3. Serwisy

Niektóre serwisy są wbudowane w proces STAFProc, inne mogą zostać załadowane jako biblioteki dynamiczne. Podobnie jak programy klienckie korzystające ze STAF, serwisy mogą być napisane w jednym z wielu języków programowania, m.in. C, C++, Java, Python, REXX.

Napisanych zostało już wiele serwisów, tak więc stworzenie własnego może nie być konieczne. Razem z dystrybucją STAF dostarczane są m.in. gotowe serwisy do:

- PROCESS: tworzenia i zarządzania procesami,
- VAR: definiowania zmiennych, używanych także przez inne serwisy,

- SEM: synchronizacji za pomocą semaforów i sekcji krytycznych,
- RESPOOL: definiowania puli zasobów, które mogą być pobierane przez aplikacje na wyłączny użytek. Przykładami zasobów, które można przydzielać z puli, są komputery testujące, adresy IP, czy licencje komercyjnych programów.
- FS: służy do wykonywania operacji na systemie plików (kopiowanie plików, tworzenie i przeszukiwanie katalogów itd),
- LOG: zapisywania informacji o wykonanych zadaniach.

Na stronie projektu dostępne są także dodatkowe serwisy: Event, Cron, Email, HTTP, Timer, ZIP.

Szczególnie rozbudowanym i ciekawym serwisem jest STAX, którego opis znajduje się w następnych punktach.

4.2. STAX

STAX, czyli *STAf eXecution engine*, jest to środowisko wykonania dla STAF. Pomaga ono zautomatyzować proces wykonania i analizy rezultatów testów, a także rozsyłania danych początkowych i zbierania wyników.

STAX oparty jest na trzech głównych technologiach:

- STAF,
- XML,
- Python.

STAX jest serwisem do STAF, stąd wszelka komunikacja z nim następuje za pomocą opisanego już mechanizmu tekstowych żądań.

Główną funkcją STAX jest wykonywanie zadań (*jobs*). Zadania te opisane są plikami XML o specjalnej strukturze, zawierającymi wstawki w języku Python[1]. Dokładniejszy opis pliku zadania przedstawiony jest w następnym punkcie.

Wraz ze STAX dostarczany jest graficzny interfejs użytkownika *STAXMon*. Pozwala on w wygodny sposób monitorować i kontrolować stan wykonujących się zadań.

4.2.1. Opis zadania

Zadania STAX przypominają tradycyjne programy lub skrypty, tyle że zapisane za pomocą składni XML. Wykonując polecenie `STAF local STAX GET DTD`, można pobrać dokument definiujący składnię dla plików z zadaniami (*DTD*). Definicję taką można wykorzystać m.in. w edytorach XML automatycznie podpowiadających dostępne elementy języka, a także do zastosowania przekształceń XSLT. Do STAX dołączone jest przykładowe przekształcenie XSLT generujące plik HTML z dokumentacją wszystkich funkcji zdefiniownych w zadaniu.

Opis zadania składa się ze znaczników XML oznaczających różne elementy języka. Za pomocą znaczników definiowana jest zarówno struktura zadania (funkcje, bloki), jak i instrukcje do wykonania.

[1] dokładnie jest to implementacja Pythona w Javie, czyli Jython

Wstawki w języku Python — <script>

Podstawową instrukcją w zadaniach STAX jest znacznik <script>. Tekst zawarty pomiędzy znacznikami <script> przekazany jest do wykonania interpreterowi języka Python. Bloki <script> służą głównie do definicji zmiennych, ale mogą być dowolnie skomplikowane i zawierać całą logikę zadania.

Poza <script>, wyrażeń Pythona można użyć w prawie wszystkich atrybutach znaczników. Wartości atrybutów są przed użyciem wyliczane przez interpreter Pythona.

Grupy instrukcji — <sequence>, <parallel>

W miejscu, w którym spodziewana jest instrukcja, może pojawić się znacznik grupujący. Wszystkie instrukcje w treści znacznika <sequence> wykonane zostaną sekwencyjnie. Jest to konstrukcja analogiczna do instrukcji złożonej w tradycyjnych językach programowania.

Ciekawym znacznikiem grupującym jest <parallel>. Podobnie jak w <sequence>, znacznik ten wykonuje wszystkie instrukcje zawarte w swojej treści, z tym że robi to równolegle, w różnych wątkach. Cały blok <parallel> kończy wykonanie dopiero wtedy, gdy zakończą się wszystkie instrukcje w nim zawarte.

Warunki, pętle

Oczywiście w języku zadań STAX możliwe jest używanie pętli i instrukcji warunkowych.

Do tworzenia pętli służy znacznik <loop>, który może pełnić funkcję dowolnej z konstrukcji *for*, *while* i *repeat ... until*, znanych z innych języków programowania. Warunki pętli podawane są za pomocą atrybutów znacznika <loop>, natomiast instrukcja do wykonania jest zawarta w jego treści.

Instrukcje warunkowe tworzy się za pomocą znaczników <if expr="warunek">, <elseif expr="warunek"> oraz <else>.

Iteracja

Aby ułatwić przechodzenie po zmiennych zawierających listy i słowniki Pythona, stworzone zostały znaczniki <iterate> oraz <paralleliterate>. Powodują one wykonanie instrukcji będącej treścią znacznika dla wszystkich elementów listy. Lista podawana jest przez atrybut i wyliczana jako wyrażenie Pythona (<iterate var="zmienna" in="wyrażenie">).

W wersji <iterate> instrukcja z treści znacznika zostanie wykonana sekwencyjnie, natomiast w <paralleliterate> równolegle.

Akcje

Ponieważ STAX oparty jest na STAF, możliwe jest korzystanie z innych serwisów tego środowiska. Służą do tego znaczniki:

- <process> — tworzy nowy proces na podanym komputerze, korzystając z serwisu PROCESS,
- <stafcmd> — pozwala wykonać dowolne żądanie STAF na wskazanym komputerze,
- <job> — uruchamia nowe podzadanie STAX.

Funkcje

W zadaniach STAX można definiować funkcje. Służą do tego znaczniki:

- `<function name="..." scope="...">` — definicja nowej funkcji,
- `<call function="..."> argumenty... </call>` — wywołanie funkcji.

Funkcje mogą przyjmować argumenty. Ich liczba oraz sposób przekazywania określone są odpowiednimi znacznikami XML: `<function-list-args>`, `<function-map-args>`, `<function-required-arg>`, `<function-optional-arg>`. Argumenty do funkcji mogą być przekazywane za pomocą listy (`<function-list-arg>` — nazwę argumentu wyznacza jego pozycja na liście) lub słownika (`<function-map-arg>` — nazwy argumentów podane są jako klucze słownika). Funkcje mogą także przekazywać wartość powrotną, korzystając ze znacznika `<return>`. Wartość przekazana w ten sposób przez funkcję dostępna jest następnie w specjalnej zmiennej Pythona o nazwie `STAXResult`.

Podczas definicji funkcji można wybrać, czy utworzone w niej zmienne będą widoczne po powrocie z tej funkcji, czy zostaną usunięte. Widocznością zmiennych lokalnych steruje argument `scope`.

Co ciekawe, znacznik `<call>` traktuje atrybut `function`, oznaczający nazwę funkcji do wywołania, jako wyrażenie Pythona. Dzięki temu możliwe jest wywołanie funkcji o nazwie wyliczonej dopiero w czasie wykonania.

Bloki opakowujące

W STAX zdefiniowane są także znaczniki modyfikujące zachowanie instrukcji zawartych w ich treści:

- `<testcase name="...">`, `<tcstatus result="pass|fail">` — oznaczają granice poszczególnych przypadków testowych i zliczają liczbę sukcesów/porażek.
- `<block>` — definiuje blok instrukcji, których wykonanie może być kontrolowane. Blok można wstrzymać, przywrócić jego wykonanie oraz zakończyć. Zarządzać blokami można z zewnątrz zadania, za pomocą żądań serwisu STAX, lub korzystając ze znaczników `<hold>`, `<release>` i `<terminate>`.

  Poza tym nazwa aktualnie wykonywanego bloku pozwala zorientować się w stanie zaawansowania zadania.
- `<timer duration="...">` — ogranicza maksymalny czas wykonywania instrukcji zawartych wewnątrz znacznika. Po przekroczeniu podanego czasu, blok zostanie przerwany, a zmienna `RC` zawiera zakodowany powód zakończenia bloku.

Inne znaczniki

Powyższa lista zawiera tylko kilka najbardziej użytecznych typów znaczników. Dostępne są również m.in. znaczniki wykorzystywane do:

- obsługi wyjątków: `<try>`, `<catch>`, `<throw>`, `<rethrow>`,
- zgłaszania sygnałów informujących o wystąpieniu asynchronicznego zdarzenia: `<raise>`, `<signalhandler>`,
- wypisywania komunikatów: `<log>`, `<message>`

STAX pozwala rozszerzyć listę obsługiwanych znaczników — każdy może stworzyć rozszerzenie obsługujące własne znaczniki.

Dołączanie bibliotek

Istotną z punktu widzenia tworzonej biblioteki cechą STAX jest możliwość dołączania definicji funkcji z zewnętrznych plików XML. Do tego celu służy znacznik `<import>`, który jako atrybut pobiera nazwę pliku XML do dołączenia do bieżącego zadania. W efekcie wykonania tego znacznika wszystkie funkcje zdefiniowane w podanym pliku będą mogły być wywołane w zadaniu.

Dzięki tej funkcjonalności możliwe jest tworzenie bibliotek używanych w różnych zadaniach STAX.

4.2.2. Przykładowe zadanie

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE stax SYSTEM "stax.dtd">

<stax>
    <defaultcall function="funkcja">
        { 'programy': [ 'ls', 'echo' ] }
    </defaultcall>

    <function name="funkcja" scope="local">
        <function-map-args>
            <function-required-arg name="programy"/>
            <function-optional-arg name="napis" default="'costam'" />
        </function-map-args>
        <sequence>
            <log>'Uruchamiam %d programów' % len(programy)</log>
            <timer duration="'15s'">
                <paralleliterate var="program" in="programy">
                    <process>
                        <location>'local'</location>
                        <command>program</command>
                        <parms>'--opcja=%s' % napis</parms>
                        <returnstdout/>
                    </process>
                </paralleliterate>
            </timer>
            <log>'Koniec'</log>
        </sequence>
    </function>
</stax>
```

4.2.3. Podsumowanie

Środowisko STAX udostępnia w miarę wygodny język do definiowania zadań. Poza zwykłymi konstrukcjami znanymi z tradycyjnych języków skryptowych, wbudowane w język zostały proste mechanizmy równoległości (`<parallel>`, `<paralleliterate>`) oraz kontroli upływu czasu (`<timer>`). Są to elementy bardzo użyteczne podczas pisania testów.

Wykorzystanie języka Python do wyliczania wyrażeń umożliwia m.in. łatwe wykonywanie operacji na napisach oraz skorzystanie z jednej z wielu gotowych bibliotek dostępnych dla tego języka. Ponadto, osobom znającym Python wystarczy bardzo niewiele czasu na naukę języka definicji zadań.

Rozdział 5

Biblioteka do testów

W tym rozdziale przedstawiona została biblioteka do wprowadzania awarii w scenariuszach testów, będąca przedmiotem niniejszej pracy.

Biblioteka zaimplementowana została jako zestaw funkcji w STAX, aby umożliwić łatwe wprowadzanie usterek w testach napisanych w tym środowisku.

5.1. Zaimplementowane mechanizmy wprowadzania usterek

W bibliotece zaimplementowane zostały następujące mechanizmy wprowadzania usterek:

Symulacja awarii zasilania:

- Wymuszenie zakończenia procesu przez wysłanie sygnału `SIGKILL`.
- Restart systemu bez zapisania buforów (`reboot -nf`, p. 3.8.2).

Blokowanie komunikacji sieciowej:

- Blokowanie pakietów o określonym docelowym adresie IP, nadawanych przez procesy użytkownika uruchamiającego test (`uid-owner`, p. 3.6.1).
- Blokowanie pakietów o określonym docelowym adresie IP, wysyłanych przez procesy użytkownika uruchamiającego test (`uid-owner`) i należących do scenariusza testowego (`gid-owner`, p. 3.6.4).
- Przypisanie testowanych procesów do puli portów (p. 3.6.9).
- Przypisanie testowanych procesów do adresów IP (p. 3.6.10), w wersji dla sieci lokalnej.

Poza porzucaniem pakietów za pomocą celu DROP możliwe jest też odpowiadanie komunikatem ICMP o błędzie (cel REJECT). Pozwala to szybciej zerwać połączenia TCP, gdy scenariusz testu tego wymaga.

5.2. Instalacja i konfiguracja biblioteki

Biblioteka do działania wymaga zainstalowanego środowiska STAF wraz z serwisem STAX. Według intencji twórców, na jednym komputerze miał działać jeden serwer STAFProc. Jednak aby odizolować od siebie działania różnych użytkowników, konfiguracją zalecaną do testów jest pojedynczy serwer STAF dla każdego użytkownika oraz ewentualnie dodatkowy serwer nadzorujący przydział wspólnych zasobów.

Od wersji 3.0 można tworzyć wiele instancji serwera STAFProc na jednej maszynie ustawiając różne wartości zmiennej środowiska `STAF_INSTANCE_NAME`. W wersjach 2.x konieczna była w tym celu modyfikacja źródeł.

Do uruchomienia serwera STAF i zaczekania na jego gotowość można posłużyć się skryptem `startSTAF.sh` dołączonym do pracy. Skrypt ten dodatkowo ustawia kilka zmiennych potrzebnych do działania biblioteki. Dodatkowa konfiguracja zależy od mechanizmów, z których użytkownik biblioteki chce korzystać.

5.2.1. Wiele zestawów testów — numer GID

Aby umożliwić równoczesne działanie wielu zestawów testów, konieczne jest uruchamianie procesów każdego zestawu z innym numerem grupy (GID).

Numery grup, z których mogą korzystać scenariusze testów, muszą zostać zdefiniowane w puli zasobów serwisu `RESPOOL` na centralnym serwerze nadzorującym wszystkie testy. Domyślna nazwa puli to `testLib-gids-`*uid*, gdzie *uid* to identyfikator użytkownika uruchamiającego test. Identyfikatory GID dla testów jednego użytkownika muszą być niepowtarzalne w całym systemie, dlatego konieczny jest centralny serwer STAF zajmujący się przydziałem identyfikatorów.

Aby utworzyć pulę i dodać do niej numery grup, można posłużyć się poleceniem:

```
STAF PoolServer RESPOOL CREATE POOL testLib-gids-1023 \
    DESCRIPTION "Pula identyfikatorów GIDs dla użytkownika o UID 1023"
STAF PoolServer RESPOOL ADD POOL testLib-gids-1023 \
    ENTRY 50000 ENTRY 50001 ENTRY 50002
```

5.2.2. Pule portów

Jeśli w testach wykorzystywana będzie technika przydzielania procesom numerów portów, to należy zdefiniować dostępne zakresy w puli zasobów. Każdy zakres powinien mieć format `port1-port2` — oznacza to odpowiednio dolne i górne ograniczenie przedziału.

Każdy komputer należący do środowiska testów powinien mieć jedną pulę portów. Domyślna nazwa puli to `testLib-portRanges-`*adres*, gdzie *adres* to główny adres IP węzła.

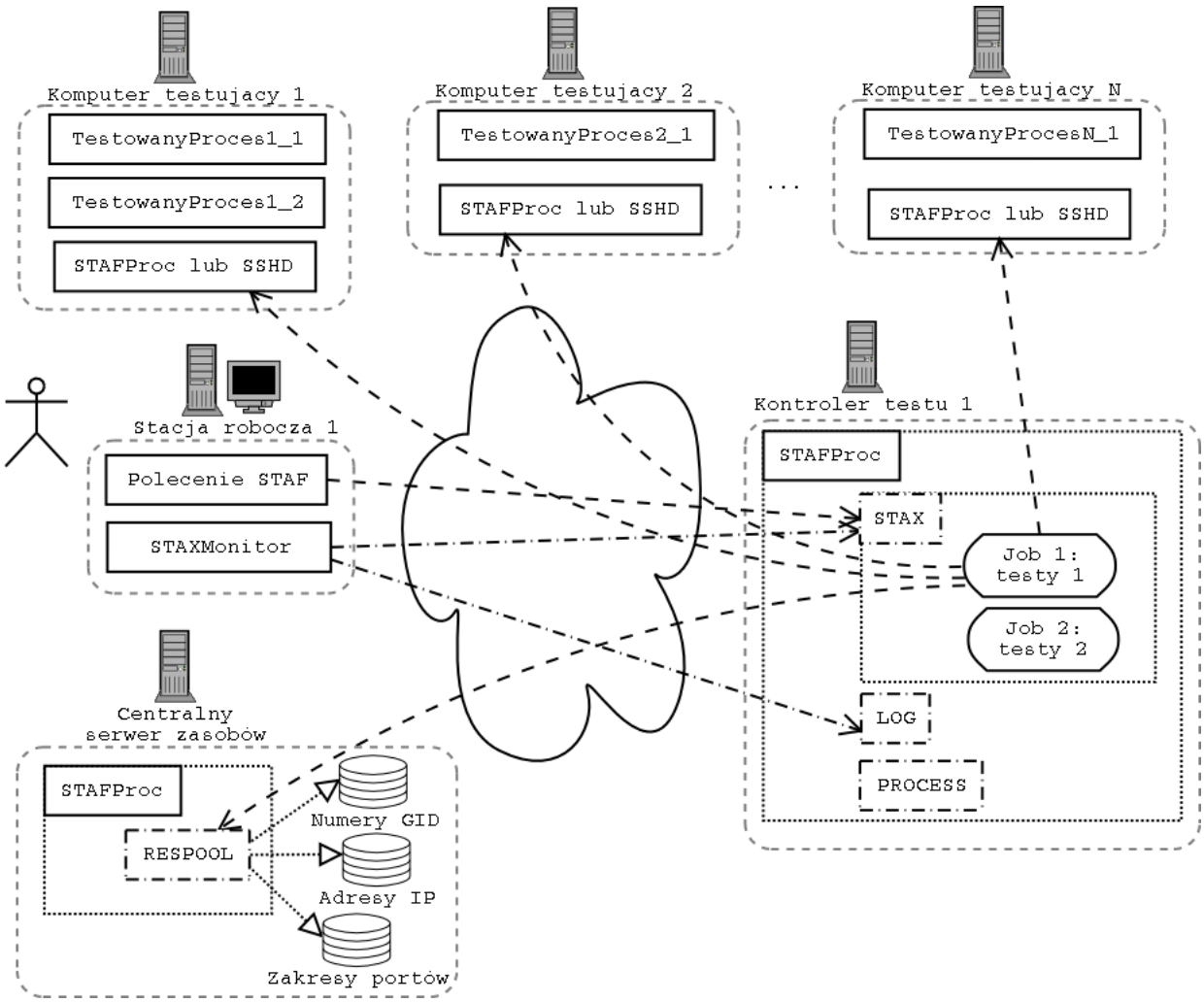
```
STAF PoolServer RESPOOL \
    CREATE POOL testLib-portRanges-192.168.7.101 \
    DESCRIPTION "Pula portów dla węzła 192.168.7.101"
STAF PoolServer RESPOOL ADD POOL testLib-portRanges-192.168.7.101 \
    ENTRY 20000-20099 ENTRY 20100-20199
```

5.2.3. Pule sztucznych adresów IP

Jeśli testowanym procesom będą przydzielane różne adresy IP, to trzeba te adresy zdefiniować w puli `testLib-ipAddrs`. Zdefiniowana pula jest wspólna dla wszystkich komputerów testujących i użytkowników.

```
STAF PoolServer RESPOOL CREATE POOL testLib-ipAddrs \
    DESCRIPTION "Pula adresów IP"
STAF PoolServer RESPOOL ADD POOL testLib-ipAddrs \
    ENTRY 172.16.104.1 ENTRY 172.16.104.2
```

5.3. Przykładowa architektura



Rysunek 5.1: Przykładowa architektura środowiska testów

Na rysunku 5.1 przedstawiono przykładową architekturę środowiska testów. W środowisku tym działają następujące komputery:

- *Komputer testujący 1–N* — maszyny, na których uruchamiane są testowane procesy,
- *Stacja robocza 1*, z której użytkownik uruchamia zestaw testów i ewentualnie śledzi jego wykonanie,
- *Kontroler testu 1*, na którym uruchomione jest zadanie STAX wykonujące scenariusz testu,
- *Centralny serwer zasobów* z uruchomionym serwisem `RESPOOL`, przydzielającym zasoby działającym zestawom testów (adresy IP, zakresy portów, identyfikatory grup).

Wszystkie te komputery połączone są siecią. Komunikację między działającymi procesami oznaczono na rysunku strzałkami.

Biblioteka testująca działa w ramach zadań STAX. Uruchomienie pliku XML z testem (lub zestawem powiązanych testów) powoduje utworzenie zadania, które wykonuje kolejne operacje zapisane w scenariuszu. Działające zadanie może komunikować się z innymi komputerami, np. uruchamiając procesy. Równocześnie w systemie może działać kilka testów, o ile używają one innych komputerów testujących lub różnych identyfikatorów GID.

Poszczególne instancje biblioteki testującej związane są z zadaniami STAX — stan biblioteki przechowywany jest wewnątrz zadania. Różne instancje biblioteki działające równocześnie w różnych zadaniach nie komunikują się między sobą bezpośrednio, jedynie korzystają ze wspólnej puli zasobów dla uniknięcia konfliktów.

Zadanie STAX wykonujące scenariusz testu musi mieć możliwość uruchamiania i kontrolowania procesów na maszynach testujących. Konkretna metoda uruchamiania procesów nie jest narzucona przez bibliotekę i może to być m.in. skorzystanie z serwisu PROCESS lub z powłoki SSH. Niezależnie od wybranej techniki, na komputerach testujących musi działać jakiś proces służący do zdalnego uruchamiania procesów, który na rysunku oznaczony został „*STAFProc lub SSHD*".

Przedstawiona architektura nie jest jedyną możliwą. Jeden komputer może łączyć kilka funkcji, np. równocześnie być kontrolerem testu, centralnym serwerem zasobów i stacją roboczą. Ponadto równocześnie w systemie może działać kilku użytkowników uruchamiających własne zadania testujące, co dla zwięzłości nie zostało uwzględnione na rysunku.

5.4. Udostępniany interfejs

Interfejs biblioteki jest zorganizowany w formie funkcji STAX. Aby w definicji testu skorzystać z blokowania komunikacji, konieczne jest dołączenie tych funkcji:

```
<import machine="'local'" file="'/sciezka/do/biblioteki/testLib.xml'">
```

Nazwy wszystkich publicznych funkcji biblioteki mają przedrostek `testLib`. Z kolei funkcje pomocnicze, które nie powinny być bezpośrednio używane z testu zaczynają się od `testLibImpl`. Jest to tylko konwencja, ale pomaga ona uniknąć konfliktów nazw funkcji między biblioteką do wprowadzania awarii a scenariuszem testu.

5.4.1. Inicjowanie biblioteki

Na początku testu konieczne jest skonfigurowanie biblioteki i poinformowanie jej o wykorzystywanych komputerach. W tym celu należy wywołać funkcję `testLib.initialize()`. Jako argumenty tej funkcji podawane są domyślne wartości konfiguracji, wspólne dla wszystkich węzłów.

Poszczególne komputery dodawane są za pomocą funkcji `testLib.addHost()`. Z każdą definicją węzła należy podać jego unikatową nazwę, główny adres IP oraz typ rozróżniania procesów na tym węźle:

- Brak — procesy uruchomione na tym węźle nie będą blokowane niezależnie.
- Unikatowy GID dla scenariusza — procesy będą uruchamiane z identyfikatorem grupy specyficznym dla zestawu testów. Procesy nie związane z testem będą działały bez przeszkód. Konieczne jest zdefiniowanie puli identyfikatorów GID.
- Własne zakresy portów dla każdego procesu — procesy uruchamiane na tym węźle będą miały przydzielone rozłączne zakresy portów. Konieczne jest zdefiniowanie puli tych zakresów.

- Unikatowe adresy IP dla procesów — każdy proces należący do testu będzie korzystał z własnego, przydzielonego dynamicznie adresu IP. Trzeba zdefiniować pulę dostępnych adresów IP.

5.4.2. Uruchamianie testowanych procesów

Stosowane techniki wprowadzania awarii wymagają ingerencji w proces uruchamiania aplikacji. Konieczne jest:

- powiadomienie biblioteki o tworzonym procesie,
- ustalenie identyfikatora dla procesu w celu późniejszej komunikacji z biblioteką,
- uruchomienie programów opakowujących proces i zmieniających jego właściwości, np. identyfikator grupy GID,
- dopisanie odpowiednich bibliotek do zmiennej `LD_PRELOAD`,
- modyfikacja zmiennych środowiska służących do konfiguracji tych bibliotek.

Dlatego wszystkie procesy uruchamiane w ramach testu powinny być zarejestrowane w bibliotece. Służy do tego funkcja `testLib.createProcess()`, którą należy wykonać przed uruchomieniem procesu. Jako parametry funkcja ta przyjmuje:

- unikatowy identyfikator procesu, nadawany przez scenariusz testu,
- identyfikator komputera, na którym ma być uruchomiony proces.

Funkcja ta w wyniku przekazuje:

- identyfikator grupy procesów, w jakiej znajdzie się proces (p. 5.5.2),
- polecenie, które należy uruchomić przed testowanym programem,
- listę zmiennych środowiska, które należy ustawić przed uruchomieniem programu.

Po uruchomieniu procesu należy poinformować bibliotekę o przydzielonym mu przez system operacyjny numerze PID, wywołując funkcję `testLib.setProcessInfo()`. Natomiast po jego zakończeniu można wykonać `testLib.removeProcess()`, aby zwolnić używane przez ten proces zasoby (zakres portów, adres IP).

Zaskakujące może się wydawać to, że biblioteka sama nie uruchamia testowanych procesów. Jednak pozostawienie tego zadania scenariuszowi testu ma szereg zalet:

- Sposób komunikacji ze zdalnymi komputerami i uruchamiania na nich programów jest często specyficzny dla zestawu testów. Można w tym celu używać choćby serwisu STAF PROCESS albo własnych połączeń SSH. Dla biblioteki testującej szczegóły tego procesu nie są istotne.
- Do korzystania z biblioteki da się łatwo zmodyfikować istniejące testy.
- Jeśli zadeklarowanie procesu i jego uruchomienie są rozdzielone w czasie, to scenariusz testu może zlecić blokadę komunikacji nowego procesu jeszcze przed jego uruchomieniem.

5.4.3. Interfejs wymagany przez bibliotekę

Do poprawnego działania, biblioteka musi uruchamiać własne programy na komputerach biorących udział w teście. Ponieważ sama nie zajmuje się wykonywaniem programów, musi skorzystać z interfejsu udostępnianego przez zestaw testów.

W tym celu wywołuje funkcję, której nazwa przekazana została podczas inicjowania biblioteki (domyślnie jest to `remoteShellExec()`). Parametry tej funkcji to:

- identyfikator komputera, na którym ma zostać uruchomione polecenie,
- polecenie do wykonania,
- flaga określająca czy polecenie ma być wykonane na prawach użytkownika systemowego czy tego, który uruchomił test,
- opcjonalnie maksymalny czas oczekiwania na zakończenie polecenia.

5.4.4. Wymuszenie zakończenia procesu

Do nagłego zakończenia procesu służy funkcja `testLib.killProcess()`. Jej parametrem jest identyfikator procesu.

Działanie funkcji polega po prostu na wysłaniu do podanego procesu sygnału SIGKILL. Oczywiście, biblioteka musiała zostać wcześniej poinformowana o numerze PID tego procesu.

5.4.5. Restart komputera

Aby zrestartować komputer testujący należy wywołać `testLib.uncleanRestart()`, jako argument podając identyfikator komputera.

5.4.6. Blokowanie komunikacji

Do blokowania komunikacji służą funkcje:

- `testLib.addNetworkPartition(zbiorProc1, zbiorProc2, typ)` — blokuje komunikację między dwoma zbiorami węzłów `zbiorProc1` i `zbiorProc2`. Sposób blokowania określony jest parametrem `typ` i może to być `"REJECT"` lub `"DROP"`.
- `testLib.removeNetworkPartition(zbiorProc1, zbiorProc2)` — funkcja odwrotna do `testLib.addNetworkPartition()`. Po jej wykonaniu wszystkie procesy ze zbioru `zbiorProc1` będą mogły się komunikować z procesami ze zbioru `zbiorProc2`.
- `testLib.disableCommunicationInSet(zbiorProc)` — blokuje łączność między wszystkimi parami procesów w zbiorze `zbiorProc`.
- `testLib.allowCommunicationInSet(zbiorProc)` — przywraca łączność między wszystkimi procesami w zbiorze `zbiorProc`.

Wywołanie tych funkcji samo nie powoduje żadnych zmian w blokadach komunikacji — ich listę należy przekazać jako argument do `testLib.modifyCommBlocking()`. Funkcje znajdujące się na liście wykonywane są sekwencyjnie i służą jedynie do zdefiniowania pożądanego stanu blokad. Zmiany w łączności sieciowej konieczne do otrzymania stanu zdefiniowanego przez listę funkcji wykonywane są jednocześnie. Dzięki temu pożądany stan blokad komunikacji można wygodnie zdefiniować za pomocą składania prostych operacji, a biblioteka sama wybierze najlepszy sposób dojścia do tego stanu.

5.5. Szczegóły realizacji biblioteki

5.5.1. Programy pomocnicze opakowujące testowane procesy

Do poprawnego działania biblioteki konieczne było utworzenie programów opakowujących testowane procesy. Działanie tych programów opisane zostało w tym punkcie.

chbindwrapper

Jest to biblioteka dynamiczna ładowana poprzez mechanizm LD_PRELOAD. Modyfikuje działanie procesu, wymuszając korzystanie przez niego z konkretnego adresu IP. Wykorzystuje technikę opisaną w rozdziale 3.6.10.

Używany adres IP oraz nazwa węzła ustawiane są na podstawie zmiennych środowiska:

`FORCE_BIND_IP`: adres IP w standardowej notacji,

`FORCE_HOSTNAME`: nazwa maszyny (powinna być tożsama z podanym adresem IP).

Kod źródłowy programu znajduje się w pliku `chbindwrapper.c`.

Przykład użycia:

```
export FORCE_BIND_IP=172.16.0.1
export FORCE_HOSTNAME=$FORCE_BIND_IP
LD_PRELOAD=/sciezka/do/biblioteki/libchbindwrapper.so ./program
```

portpoolwrapper

Jest to także biblioteka dynamiczna ładowana mechanizmem LD_PRELOAD. Wymusza korzystanie przez proces z ustalonej puli portów, zgodnie z opisem w rozdziale 3.6.9.

Konfiguracja puli portów odbywa się za pomocą zmiennych środowiska:

`FORCE_BIND_PORT_RANGE`: zakres portów, których proces może użyć w wywołaniach `bind()`. Format: dwie liczby oznaczające początek i koniec zakresu.

`FORCE_DYNAMIC_PORT_RANGE`: zakres portów, które będą przydzielane automatycznie przez bibliotekę, jeśli proces nie wykona `bind()`.

Krótkiego wyjaśnienia wymaga użycie dwóch zakresów portów zamiast jednego. Gdy testowane procesy uruchamiane są równocześnie przez kilku użytkowników, konieczne jest zróżnicowanie używanych przez nich numerów portów. Tradycyjnie problem ten rozwiązuje się przydzielając każdemu programiście jego własny, niewielki zakres portów. Zakres ten jest używany wyłącznie dla gniazd o znanych numerach, na których proces nasłuchuje, aby możliwe było nawiązanie z nim pierwszego połączenia.

Istotną cechą tych zakresów jest to, że są one rozłączne i stałe dla każdego programisty. Jest to wygodne np. ze względu na utrzymanie stałej konfiguracji przez każdego z użytkowników. Jednak oznacza to, że zakresy te nie mogą być duże. Gdyby przydzielane z nich były także porty dynamiczne, zakres przypisany użytkownikowi szybko by się wyczerpał. Ponieważ wszyscy programiści nie korzystają naraz z jednego komputera, można zdefiniować większe zakresy portów dynamicznych, które przydzielane będą tylko użytkownikom aktywnie korzystającym z konkretnego węzła.

Dzięki zastosowaniu dwóch przedziałów, porty nasłuchujące pozostają przydzielone statycznie, a nieistotne porty połączeń wychodzących wybierane są z dużej puli portów dynamicznych.

Kod źródłowy programu znajduje się w pliku `portpoolwrapper.c`.
Przykład użycia:

```
export FORCE_BIND_PORT_RANGE="9000 9009"
export FORCE_DYNAMIC_PORT_RANGE="20000 20999"
LD_PRELOAD=/sciezka/do/biblioteki/libportpoolwrapper.so ./program
```

5.5.2. Implementacja blokowania komunikacji

W tym punkcie opisana została implementacja blokowania komunikacji, będąca najwiekszą częścią biblioteki do wprowadzana awarii.

Zasada działania

Na wysokim poziomie, biblioteka ma następujące zadania:

- zbiera informacje o tworzonych i kończonych procesach,
- modyfikuje procedurę uruchamiania testowanych procesów tak, aby wykonane zostały odpowiednie programy i biblioteki opakowujące,
- tłumaczy wysokopoziomowe żądania użytkownika („zablokuj łączność między podanymi zbiorami procesów") na polecenia modyfikujące reguły iptables,
- pamięta i uaktualnia bieżący stan blokad komunikacji,
- weryfikuje poprawność żądań użytkownika, np. czy dwa procesy uruchomione na jednym węźle nie są blokowane niezależnie, gdy nie zostały dla nich zdefiniowane żadne zakresy portów ani unikatowe adresy IP.

W dalszej części opisane zostaną dokładniej operacje, które musi wykonać biblioteka by zapewnić tę funkcjonalność.

Grupy procesów

Aby w sposób abstrakcyjny wyrazić zależności między procesami uruchomionymi na jednym węźle, wprowadzone zostało pojęcie *grupy procesów*. Jest to najmniejsza jednostka, dla której możliwe jest sterowanie blokadami komunikacji. Wszystkie procesy w grupie mogą się komunikować między sobą.

Jako pierwszy element przetwarzania przez bibliotekę, żądania użytkownika dotyczące poszczególnych procesów tłumaczone są na operacje na grupach. Jeśli w wyniku tych operacji procesy należące do jednej grupy będą mogły komunikować się z różnymi zbiorami procesów zdalnych, to zgłoszony zostanie błąd.

Po tej fazie całe przetwarzanie odbywa się już na poziomie grup, a nie pojedynczych procesów.

Stan blokad

Aby efektywnie zarządzać wprowadzonymi blokadami, konieczna jest znajomość aktualnie wprowadzonych reguł porzucających pakiety. Konieczne jest więc pamiętanie stanu.

Cały stan biblioteki pamiętany jest w globalnych zmiennych STAX. Zmienne globalne STAX, w przeciwieństwie do zwykłych zmiennych Pythona, nie są kopiowane podczas uruchamiania nowych wątków i wywoływania funkcji. Do ich definicji służy specjalny obiekt `STAXGlobal`.

Zdefiniowane są dwie takie zmienne:

- `GLOBAL_commBlockState`: stan biblioteki. Jest to obiekt klasy zawierającej:
  - translację z identyfikatora procesu na identyfikator grupy procesów, do której należy ten proces,
  - informacje o każdej grupie procesów takie jak: adres IP, zakresy portów, identyfikator GID oraz węzeł, na którym uruchomione są procesy tej grupy,
  - informacje o wszystkich aktualnie wprowadzonych blokadach wraz z ich typem (`REJECT` czy `DROP`).
- `GLOBAL_commBlockConfig`: konfiguracja biblioteki dla aktualnego zestawu testów. Nie zmienia się w trakcie trwania testu.

Uruchamianie procesów

Biblioteka zbiera informacje o uruchomionych procesach dzięki wywołaniom funkcji `testLib.createProcess()`. Z każdym procesem wiązane są pewne informacje, takie jak węzeł, na którym został uruchomiony, zakresy używanych portów oraz adresów IP (jeśli włączone zostały odpowiednie biblioteki opakowujące).

Podczas uruchamiania, proces przydzielany jest do odpowiedniej grupy. Jeśli konfiguracja węzła umożliwia blokowanie na poziomie poszczególnych procesów, to proces sam stanowi swoją grupę. W przeciwnym razie wszystkie procesy uruchamiane na tym węźle trafiają do wspólnej grupy.

Podczas rejestrowania procesu pobierane są odpowiednie zasoby z puli — adres IP lub zakres portu. W przypadku adresu IP tworzony jest też odpowiedni alias interfejsu sieciowego, zgodnie z opisem w p. 3.6.10.

Opis algorytmu

Funkcją służącą do blokowania komunikacji przez użytkownika biblioteki jest `testLib.modifyCommBlocking()`. Jako argument dostaje ona listę operacji blokowania lub przywracania łączności sieciowej, które mają być wykonane jednocześnie.

Funkcja ta ułatwia zdefiniowanie pożądanego stanu, poprzez składanie pewnych prostych operacji — bezpośrednie manipulacje poszczególnymi połączeniami byłyby niewygodne. W wyniku wykonania wszystkich operacji przekazanych do `testLib.modifyCommBlocking()`, tworzone są dwie listy: tych połączeń (par *(nadawca, odbiorca)*), które powinny zostać zablokowane, oraz tych, które powinny zostać przywrócone. Stan łączności między węzłami nie wyspecyfikowanymi w żadnej z tych list powinien pozostać niezmieniony.

Po wyliczeniu, obie listy przekazywane są do funkcji `testLibImpl.applyCommBlockChange()`. Jest to rzeczywista implementacja blokowania komunikacji.

Pierwszym zadaniem tej funkcji jest weryfikacja zgodności żądanego stanu z ograniczeniami użytej techniki blokowania. Zajmuje się tym funkcja `testLibImpl.removeAlreadyDoneProcessBlocks()`, która jednocześnie usuwa z list te połączenia, które już znajdują się w odpowiednim stanie.

Następnie żądania dotyczące poszczególnych procesów łączone są w operacje na całych grupach (funkcja `testLibImpl.mergeProcessesToGroups()`). W wyniku otrzymywane są dwie listy:

- lista par (grupa źródłowa, grupa docelowa), dla których dotychczas komunikacja była zablokowana, a ma być przywrócona,
- lista par (grupa źródłowa, grupa docelowa), które mają zostać zablokowane odpowiednim typem blokady (REJECT lub DROP), a które dotychczas:
  - mogły się komunikować lub
  - były zablokowane drugim rodzajem blokady.

Kolejnym krokiem jest zdecydowanie, dla każdej pary (nadawca, odbiorca), po której stronie dopisane zostaną reguły iptables. Oczywiście jest to możliwe tylko dla procesów przypisanych do konkretnego adresu IP. W obecnej implementacji wybierana jest zawsze strona nadawcy — funkcja `testLibImpl.assignBlockSide()` jest zaślepką.

Po tych operacjach możliwe jest już wygenerowanie listy operacji do wykonania na każdym węźle. Każda operacja odpowiada jednej lub kilku regułom iptables, wstawianym lub usuwanym z łańcuchów jądra na odpowiednim komputerze. Przygotowaniem listy operacji dla każdego z węzłów, których dotyczy modyfikacja blokad, zajmuje się funkcja `testLibImpl.prepareHostOperationsMap()`.

Pojedyncza operacja jest abstrakcyjną reprezentacją reguły, która ma zostać dodana lub usunięta. Jej postać jest następująca:

- źródłowa grupa procesów,
- docelowa grupa procesów,
- typ blokady,
- czy reguła dotyczy pakietów przychodzących czy wychodzących.

Przygotowane w ten sposób reguły są wykonywane na przypisanych im węzłach. Rzeczywiste operacje na łańcuchach iptables na różnych węzłach odbywają się równolegle dzięki wykorzystaniu konstrukcji `<paralleliterate>`.

5.5.3. Podsumowanie i możliwe rozszerzenia

Biblioteka zapewnia wszystkie wymagane funkcje wprowadzania awarii. Interfejs biblioteki umożliwia jej użycie w różnych środowiskach testów oraz zapewnia wysoką konfigurowalność. W obecnie wykonywanych testach biblioteka działa wystarczająco dobrze, możliwe są jednak pewne usprawnienia.

W miarę prostym rozszerzeniem biblioteki byłoby zaimplementowanie algorytmu wybierającego stronę, po której blokowana jest komunikacja. Stosowanie blokad po stronie odbiorcy poprawiłoby wydajność zmian stanu blokad, szczególnie w sytuacjach, gdy od reszty systemu odłączana jest drobna grupa procesów. Cała biblioteka jest przygotowana do takiego działania, wobec tego nakład pracy nie powinien być duży.

Aby dalej zmniejszyć czas potrzebny do zmodyfikowania stanu blokad można wykorzystać niestandardowy moduł dopasowujący `condition`. Moduł ten pozwala zdefiniować zmienne sterujące dopasowywaniem reguł do pakietów — jeśli jednym z warunków reguły jest `condition`, to zostanie ona dopasowana jedynie jeśli związana z nią zmienna ma wartość

niezerową. Zmienne reprezentowane są przez pliki w systemie `/proc` a ich wartość można ustawiać prostymi zapisami. Korzystając z reguł `condition`, można więc sterować stanem blokad komunikacji bez powolnych operacji na łańcuchach iptables.

Kolejną możliwością poprawy wydajności jest ograniczenie liczby tworzonych reguł. Liczba koniecznych do wstawienia reguł iptables ogranicza skalowalność biblioteki. Istnieją np. moduły pozwalające w wydajny sposób dopasowywać pakiety o adresach należących do dynamicznie ustawianego zbioru. Niestety, moduły te nie należą do standardowej dystrybucji iptables dołączanej do Linuksa.

Nawet bez dodatkowych modułów można zmniejszyć liczbę reguł, gdy blokowanych jest wiele kolejnych adresów IP. Spójne grupy blokowanych adresów można łączyć w podsieci IP i blokować całą taką podsieć. Nieco bardziej ogólnym mechanizmem byłoby utworzenie drzewa reguł pasujących do kolejnych fragmentów przestrzeni adresów. W takim rozwiązaniu liczba reguł, przez jakie musi przejść pakiet, zależałaby od logarytmu liczby zablokowanych procesów.

Rozdział 6

Testy wydajności technik wprowadzania awarii

W tym rozdziale przedstawiam wyniki testów wydajności mechanizmów wprowadzania awarii opisanych w rozdziale 3 oraz zaimplementowanych w bibliotece testującej.

6.1. Sposób przeprowadzenia testów

6.1.1. Reguły iptables

Aby ocenić wydajność technik korzystających z iptables (p. 3.6.1) do jądra wstawiana była odpowiednia liczba reguł filtrujących pakiety. Reguły te były wstawiane do własnego łańcucha `test-iptables-overhead` i miały następującą postać:

```
iptables -A test-iptables-overhead -s $srcip -d $dstip -j DROP
```

gdzie `$srcip` i `$dstip` to adresy IP z pewnej nieużywanej sieci. Z kolei `test-iptables-overhead` wywoływany był ze standardowego łańcucha OUTPUT po dopasowaniu identyfikatorów `owner-uid` i `owner-gid`. W ten sposób pakiety przechodziły przez układ reguł taki sam jak w bibliotece testującej, a jednocześnie możliwa była dokładna kontrola nad liczbą tych reguł. Pakiety testowanych procesów nie pasowały do żadnej z nich, więc musiały przechodzić przez cały łańcuch `test-iptables-overhead`.

6.1.2. ptrace()

Do badania narzutu spowodowanego śledzeniem za pomocą `ptrace()` wykorzystany został program `ptracer`. `Ptracer` uruchamia proces podany jako argument i, korzystając z `ptrace(PTRACE_SYSCALL)`, przechwytuje wszystkie wywołania funkcji systemowych śledzonego procesu. Po przechwyceniu wywołania funkcji systemowej, `ptracer` natychmiast kontynuuje działanie procesu.

Program `ptracer` służy do oszacowania dolnego ograniczenia narzutu wprowadzanego przez wszystkie techniki korzystające z mechanizmu `ptrace()`.

Kod źródłowy `ptracer` znajduje się na płycie CD dołączonej do pracy.

6.1.3. User-mode Linux

W testach wykorzystany został User-mode Linux 2.6.11.11 w wersji nie wymagającej żadnych łat na system gospodarza.

6.1.4. FAUmachine

Wersja FAUmachine użyta do testów to `faumachine-20050728`. System uruchamiany wewnątrz maszyny wirtualnej to zmodyfikowana wersja Linuksa 2.4.20 z dystrybucji Redhat 9, rozprowadzana wraz z pakietem FAUmachine. Także w tym przypadku jako system gospodarza zastosowano standardowe jądro, bez żadnych łat specyficznych dla FAUmachine.

Użyta wersja FAUmachine udostępnia dwie implementacje wirtualnego procesora:

- `JIT`: Programy wykonywane są na rzeczywistym procesorze. Instrukcje uprzywilejowane i odwołujące się do urządzeń są przepisywane na wywołania funkcji symulatora. Jest to mechanizm opisany w p. 2.3.2.

- `QEMU`: Instrukcje wykonywane są w symulatorze procesora x86 stworzonym w ramach projektu QEMU[11]. Maszyna wirtualna nie musi przechwytywać wywołań funkcji systemowych, za to znacznie wzrasta koszt wykonywania obliczeń, także w trybie użytkownika.

6.2. Narzut na czas obsługi pakietu

Testy opisane w tym punkcie służą do zbadania czasu obsługi pakietu IP przez system operacyjny przy różnych technikach blokowania komunikacji.

Do testów przygotowano dwa proste programy:

- `test-reply port` — nasłuchuje na podanym porcie UDP i odpowiada na każdy odebrany pakiet pakietem o takich samych danych,

- `test-send ip port num len` — wysyła pakiet UDP o długości `len` pod podany adres `ip:port` i czeka na odpowiedź. Po otrzymaniu pakietu powrotnego powtarza całą operację. Po wykonaniu `num` pełnych cykli proces wypisuje czas trwania testu i kończy działanie.

Korzystając z tych programów można zmierzyć średni czas potrzebny na przebycie przez pakiet pełnej drogi od procesu nadającego, przez system operacyjny nadawcy, ewentualnie sieć i system operacyjny odbiorcy aż do procesu odbierającego.

Aby wyeliminować błędy spowodowane obciążeniem sieci, oba procesy zostały uruchomione na tym samym komputerze.

6.2.1. Porównanie iptables, ptrace() i UML

W przeprowadzonym teście procesy przesyłały 100000 żądań (czyli 200000 pakietów) o długości 1 bajta. Oba procesy `test-send` i `test-reply` uruchomione były tej samej maszynie.

Testowy komputer był wyposażony w procesor Pentium III 850MHz oraz jądro Linux 2.4.20.

Wyniki testu przedstawia tabela 6.1.

Mechanizm wprowadzana usterek		Średni czas trwania testu [s]	Odchylenie standardowe
Brak — sam system operacyjny		2.850	0.014
Reguły iptables (tylko `test-send`)	10 reguł	2.904	0.017
	100 reguł	3.342	0.022
	200 reguł	3.810	0.016
	1000 reguł	8.92	0.211
Reguły iptables (oba procesy)	10 reguł	3.029	0.004
	100 reguł	3.865	0.009
	200 reguł	4.828	0.014
	1000 reguł	13.935	0.166
`ptracer`	tylko `test-send`	4.811	0.026
	oba procesy	6.327	0.022
User-mode Linux	(oba procesy)	80.084	2.245
FAUmachine (oba procesy)	JIT	463.768	1.044
	QEMU	236.954	0.766

Tabela 6.1: Czas obrotu 100000 pakietów UDP o długości 1 bajta

Analiza rezultatów

Test ten wykazał znaczną przewagę technik opartych na regułach iptables nad maszynami wirtualnymi i `ptrace()`.

Dla niewielkich długości łańcuchów czas przetwarzania reguł iptables jest pomijalny. Nawet dla 100 reguł, narzut wynosi zaledwie 17% minimalnego czasu obsługi pakietu przez jądro. W rzeczywistym systemie pojawiają się dodatkowo koszty obsługi pakietu przez protokół TCP oraz sterownik karty sieciowej.

W zaimplementowanych technikach blokowania komunikacji, każda reguła iptables odpowiada na ogół blokadzie jednego procesu. Przy 100 regułach oznacza to więc, że uruchomionych zostało aż 100 testowanych procesów.

Narzut wynikający z zastosowania `ptrace()` okazał się równoważny wstawieniu ok. 400 reguł iptables. Ponadto test `ptrace()` jest bardzo wyidealizowany — nie uwzględnia choćby problemu zdecydowania, czy pakiet należy porzucić oraz rozpoznawania numeru wywołanej funkcji systemowej. Jest to tylko dolne ograniczenie narzutu związanego z każdym rozwiązaniem opartym na `ptrace()`. Okazało się, że nawet to dolne ograniczenie jest większe niż kompletny koszt blokowania za pomocą reguł iptables dla sensownych liczb procesów.

Jak można się było spodziewać, bardzo źle wyszły w tym teście maszyny wirtualne UML i FAUmachine. Opóźnienia przez nie wprowadzane są o rząd wielkości większe niż te dla reguł iptables.

6.2.2. Zależność czasu obsługi pakietu od liczby reguł iptables

W wynikach poprzedniego testu (tabela 6.1) można zauważyć zależność czasu obsługi pakietu od liczby reguł iptables, przez które ten pakiet przechodzi. Kolejny test ma na celu wykrycie rodzaju tej zależności. Ponieważ biblioteka do wprowadzania awarii tworzy reguły iptables podczas blokowania komunikacji, test ten pozwoli oszacować skalowalność biblioteki.

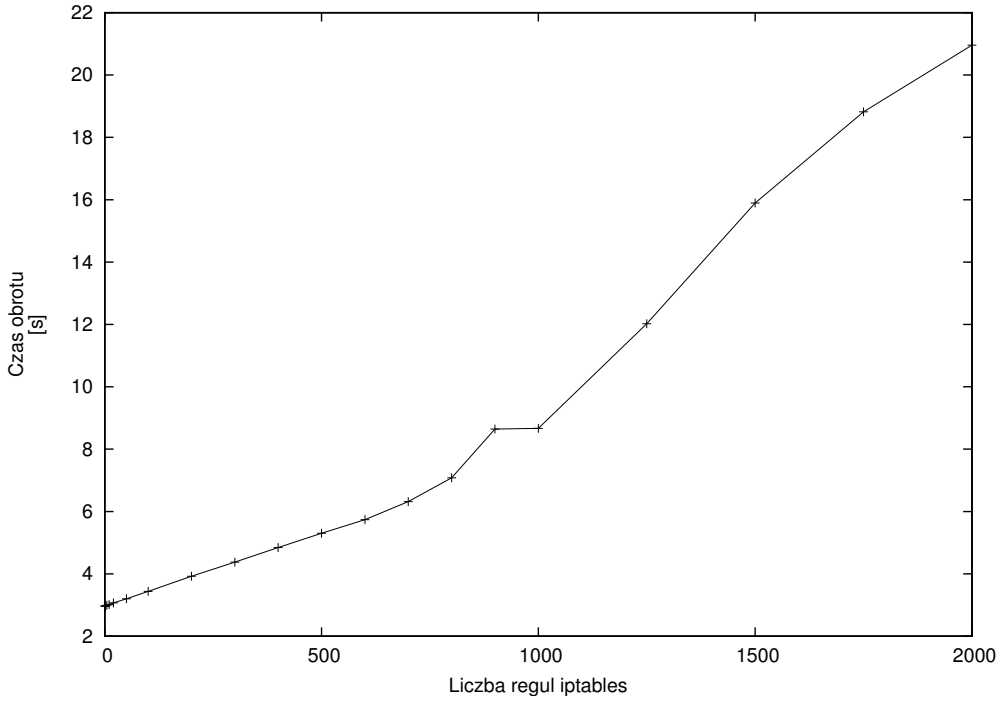
Test przeprowadzony został w takiej samej konfiguracji co poprzedni. Jeden z procesów był uruchomiony z innym identyfikatorem grupy i tylko pakiety wysyłane przez ten proces

przechodziły przez łańcuch reguł `test-iptables-overhead`. Dzięki temu narzut reguł iptables liczony był tylko raz.

Wyniki testu przedstawiono w tabeli 6.2 oraz na rys. 6.1.

Liczba reguł	0	1	2	5	10	20	50
Średni czas	2.967	2.971	2.963	2.991	3.016	3.071	3.197
Odchylenie standardowe	0.039	0.011	0.018	0.011	0.009	0.009	0.020
Liczba reguł	100	200	300	400	500	600	700
Średni czas	3.438	3.923	4.375	4.844	5.301	5.736	6.317
Odchylenie standardowe	0.004	0.019	0.012	0.017	0.007	0.040	0.042
Liczba reguł	800	900	1000	1250	1500	1750	2000
Średni czas	7.080	8.642	8.667	12.021	15.892	18.819	20.956
Odchylenie standardowe	0.081	0.095	0.099	0.177	0.176	0.101	0.067

Tabela 6.2: Czas obsługi pakietu w zależności od liczby dopasowywanych reguł iptables



Rysunek 6.1: Czas obrotu 100000 pakietów w zależności od liczby reguł

Zgodnie z przewidywaniami, czas obsługi pakietu przez jądro rośnie liniowo z liczbą dopasowywanych do niego reguł. Przy około 800 regułach zależność ta zostaje zaburzona i wzrost złożoności staje się szybszy. Prawdopodobnie wynika to z przekroczenia rozmiaru pamięci podręcznej procesora przez nadmiernie rozbudowane zestawy reguł.

6.3. Przepustowość łączy TCP

6.3.1. Przepustowość lokalnych połączeń

Celem testu jest porównanie maksymalnej przepustowości osiąganej przez procesy uruchomione na jednej maszynie, wysyłające między sobą dane przez połączenia TCP. Pozwala on ocenić rzeczywisty wpływ opóźnień i zwiększonego zużycia czasu procesora, związanych z zastosowaniem różnych metod blokowania komunikacji, na dostępną dla procesów przepustowość połączeń sieciowych.

W teście biorą udział dwa procesy:

- `band-server`: oczekuje na połączenie TCP na podanym porcie, a po jego nawiązaniu przesyła dane (ciąg bajtów o wartości 0),
- `band-client`: łączy się z podanym adresem TCP i odczytuje otrzymane dane; po zamknięciu połączenia wypisuje średnią przepustowość połączenia w KB/s.

Rozmiar bufora zapisu i odczytu procesów, czyli liczba bajtów przesyłana jednym wywołaniem `write()`, ustawiony został na 100000. Tak więc przesyłane są duże pakiety danych.

W przeprowadzonym teście przesyłanie danych trwało 25 sekund. Po tym czasie do procesu `band-server` wysyłany był sygnał SIGTERM kończący jego działanie. Do czasu przesyłania danych nie był wliczany czas nawiązywania połączenia.Ponieważ test dotyczy lokalnej wydajności komunikacji, oba procesy uruchomione zostały na tej samej maszynie.

Wyniki

Testy przeprowadzone zostały na czterech różnych komputerach:

- jednoprocesorowym Pentium III 850MHz z zainstalowanym jądrem Linux 2.4.20,
- jednoprocesorowym Celeron 2.66GHz z jądrem Linux 2.6.10,
- dwuprocesorowym Intel Xeon 2.40GHz z jądrem Linux 2.6.9,
- dwuprocesorowym AMD Athlon MP 2GHz z jądrem Linux 2.6.12.5.

Przez reguły iptables przechodziły zarówno pakiety serwera, jak i klienta.

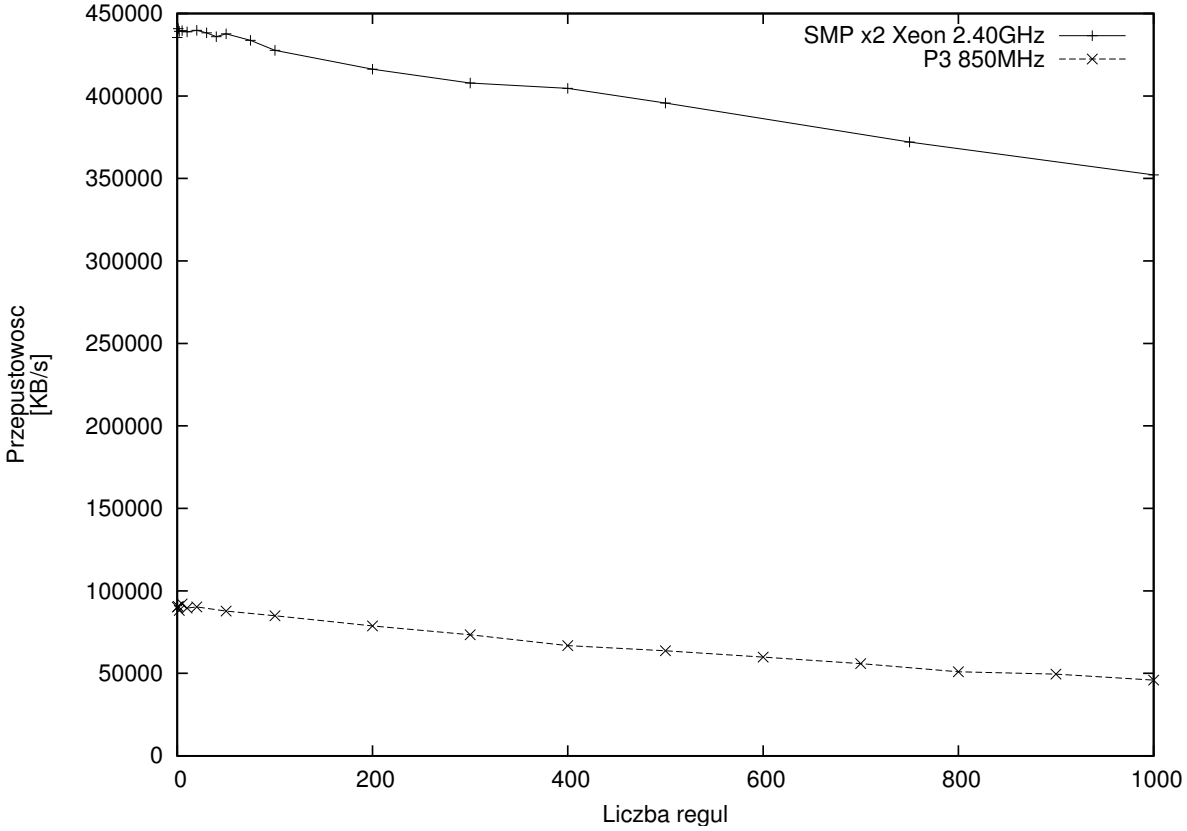
	P3 850MHz UP Linux 2.4.20	Celeron 2.66GHz UP Linux 2.6.10	Xeon 2.40GHz SMP (x2) Linux 2.6.9	Athlon MP 2GHz SMP (x2) Linux 2.6.12.5
Bazowy system	86028	193105	440682	217796
10 reguł iptables	83883	191828	438900	213661
100 reguł iptables	76394	187499	427707	207020
Oba procesy pod `ptracer`	76315	179946	450692	207856
User-mode Linux	26175	76338	76096	77812
FAUmachine JIT	5793	9014	8875	9677
FAUmachine QEMU	6331	-	-	-

Tabela 6.3: Przepustowość lokalnych połączeń TCP w KB/s

Wyniki testu przedstawione zostały w tabeli 6.3.

Test wykonywany był dziesięciokrotnie dla każdej konfiguracji komputera i mechanizmu blokowania komunikacji. Przedstawione wyniki to średnie z przepustowości osiągniętych w kolejnych uruchomieniach. Odchylenie standardowe w żadnym przypadku nie przekroczyło 3%, więc nie zostało uwzględnione w tabeli, aby nie zaciemniać jej struktury.

Dla dwóch komputerów, które uzyskały odpowiednio najwyższą i najniższą przepustowość, sporządzony został również wykres przepustowości od liczby reguł iptables (rys. 6.2).



Rysunek 6.2: Wykres zależności przepustowości lokalnych połączeń TCP od liczby reguł

Wyniki tego testu zgodne są z tymi, jakie były otrzymane w teście opóźnienia — najlepszą wydajność zapewniło blokowanie przez reguły iptables, nieznacznie gorszy (na UP) był mechanizm `ptrace()`. Nawet przy przesyłaniu dużych ilości danych, gdy koszt pojedynczego wywołania systemowego miał szansę się zamortyzować, User-mode Linux był 2,5–6 razy wolniejszy niż rzeczywisty system. FAUmachine wypadł bardzo źle, z maksymalną przepustowością lokalną rzędu 6-9 MB/s. Ponadto tylko na jednym z komputerów udało się uruchomić FAUmachine w trybie QEMU, na pozostałych zgłaszane były błędy podczas inicjowania jądra wirtualnego systemu. Przyczyna tych błędów nie jest znana.

Ciekawym wynikiem jest wzrost przepustowości na maszynie dwuprocesorowej przy wykorzystaniu `ptrace()`. Prawdopodobnie `ptracer` wymusił przypadkowo lepszy przeplot działających równocześnie procesów, zwiększając nieznacznie ich efektywność. Ponadto negatywne cechy `ptrace()` przy wielu procesorach są znacznie zmniejszone — proces śledzący może wykonywać się na innym procesorze niż śledzony, dzięki czemu nie jest konieczne przełączenie

kontekstu przy każdym budzeniu procesu `ptracer`.

Jak można było przewidywać, procesy działające pod User-mode Linuxem nie mogą skorzystać z wieloprocesorowości zewnętrznej maszyny, dlatego nie zadziałały szybciej na komputerach SMP niż na jednoprocesorowych.

6.3.2. Przepustowość połączeń TCP w sieci Gigabit Ethernet

Ten test ma za zadanie ocenić maksymalną przepustowość połączeń TCP przez gigabitową sieć Ethernet dla porównywanych technik wprowadzania usterek sieci. Działanie tego testu jest prawie identycznie jak poprzedniego, tyle tylko, że procesy uruchomione zostały na różnych maszynach, połączonych siecią gigabitową.

Przy wydajnych mechanizmach blokowania komunikacji, przepustowość połączeń przesyłających dane w rzeczywistej sieci będzie ograniczana przez możliwości sprzętu. Celem tego testu jest zbadanie, czy procesy poddane różnym technikom wprowadzania awarii będą w stanie wypełnić w całości sieć gigabitową, którą połączone są komputery testujące.

O ile dla rzeczywistego systemu sposób podłączenia do sieci jest oczywisty, o tyle User-mode Linux udostępnia kilka mechanizmów połączenia maszyn wirtualną siecią. W tym teście wykorzystano dwa z tych sposobów:

- `mcast`: pakiety maszyn wirtualnych kapsułkowane są w pakietach rzeczywistej sieci, rozsyłanych za pomocą mechanizmu multicast. Opakowane pakiety sieci wirtualnej trafiają do wszystkich User-mode Linuksów należących do tej samej grupy multicast.

- `tuntap`: maszyna wirtualna User-mode Linux połączona jest z rzeczywistą siecią poprzez odpowiednio skonfigurowany interfejs `tap` w systemie gospodarza. W przeciwieństwie do `mcast`, ten sposób wymaga dość znacznych ingerencji w konfigurację rzeczywistej sieci.

Test został przeprowadzony na dwóch identycznych maszynach dwuprocesorowych Athlon MP 2GHz z jądrem Linux 2.6.12.5 SMP.

Wyniki

Wyniki przedstawiono w tabeli 6.4.

Mechanizm blokowania komunikacji	Przepustowość w KB/s	Odchylenie standardowe
Bazowy system	114832	73
100 reguł iptables	114693	346
200 reguł iptables	112494	1061
Oba procesy uruchomione pod `ptracer`	114746	133
User-mode Linux (`mcast`)	47	2
User-mode Linux (`tuntap`)	28135	732

Tabela 6.4: Przepustowość połączeń TCP po sieci Gigabit Ethernet

Blokowanie przez reguły iptables aż do 100 reguł oraz śledzenie przez `ptrace()` nie mają żadnego wpływu na przepustowość — czynnikiem ograniczającym jest w tych przypadkach sama sieć. Oznacza to, że obie techniki pozwalają wykorzystać sieć gigabitową w pełni. Nawet przy 200 regułach iptables spadek przepustowości jest nieznaczny.

Z kolei nawet najprostsze procesy uruchomione pod User-mode Linux nie są w stanie wykorzystać pełnej przepustowości sieci gigabitowej. Pod FAUmachine nawet w przypadku lokalnym nie są w stanie przekroczyć 6-9 MB/s. Potwierdziło to przypuszczenia, że maszyny wirtualne są zbyt powolne do testowania systemu rozproszonego opisanego w tej pracy.

6.4. Wnioski z testów

Przeprowadzone testy potwierdziły przewidywania opisane w rozdziale 3.

Mechanizmem blokowania komunikacji o najmniejszym narzucie są reguły iptables. Reguły te mają niewielki wpływ na wydajność systemu, a narzut przez nie wprowadzany rośnie liniowo z liczbą reguł, przez które przechodzi pakiet. Do około 100 reguł praktycznie niezauważalny jest ich wpływ zarówno na przepustowość, jak i na opóźnienia przesyłanych danych.

Z kolei maszyny wirtualne, tal User-mode Linux, jak i FAUmachine, powodują co najmniej kilkukrotne zwiększenie narzutu. Jak wykazał ostatni test, procesy uruchomione w maszynach wirtualnych nie są w stanie nawet wykorzystać pełnej przepustowości sieci Gigabit Ethernet. Należy pamiętać, że procesy w testowanym systemie poza siecią korzystają także z dysków oraz wykonują intensywne obliczenia.

6.5. Praktyczne doświadczenia

Biblioteka do wprowadzania awarii napisana w ramach tej pracy wykorzystywana jest już w testach systemu opisanego w rozdziale 1.1. Dzięki jej użyciu udało się znaleźć i naprawić wiele błędów w rozwijanym systemie.

Wpływ biblioteki na przepustowość testowanego systemu jest niezauważalny. Obecnie, pojedynczy scenariusz testu działa na maksymalnie kilkunastu maszynach, tak więc liczba wstawianych reguł iptables jest niewielka.

Rozdział 7

Podsumowanie

W pracy przeanalizowane zostały różne mechanizmy symulowania awarii w testach systemów rozproszonych. Ze względu na charakter tych systemów, główny nacisk położono na mechanizmy wprowadzania usterek komunikacji sieciowej między współpracującymi procesami.

Największą wierność wprowadzanych usterek zapewnia użycie maszyn wirtualnych o dostępnym kodzie źródłowym. Niestety rozwiązanie to wprowadza znaczny narzut nawet w tych momentach, gdy nie są symulowane żadne usterki. Narzut ten mógł zaburzać testy wydajności systemu, więc skorzystanie z maszyn wirtualnych okazało się nieodpowiednie do przedstawionego w tej pracy typu testów.

Najlepszym sposobem symulowania awarii sieci okazało się być wykorzystanie odpowiednich reguł ściany ogniowej iptables. Rozwiązanie to wprowadza tylko niewielki narzut podczas działania blokad komunikacji. Natomiast po usunięciu wszystkich symulowanych usterek narzut znika całkowicie i testowane procesy mogą kontynuować działanie z pełną wydajnością. Ponadto blokowanie komunikacji na poziomie pojedynczych pakietów pozwala przetestować poprawność użycia interfejsów systemu operacyjnego — np. czy ustawione zostały odpowiednie opcje gniazda skracające maksymalny czas oczekiwania na odpowiedź.

Ciekawy wniosek płynący z pracy dotyczy stopnia konfigurowalności systemu Linux. Techniki wprowadzania awarii wykorzystane w stworzonej bibliotece ingerują dość znacznie w proces przetwarzania pakietów przez system operacyjny. Jednocześnie, wykorzystują jedynie mechanizmy dostępne w standardowej dystrybucji Linuksa. Okazało się, że do bardzo wielu zastosowań, nawet nietypowych, nie jest konieczne pisanie łat do Linuksa — wystarczy jedynie skorzystać z mechanizmów już obecnych w tym systemie.

Wiele decyzji projektowych podjętych podczas pracy związanych było z wymaganiami specyficznymi dla aplikacji, która będzie testowana przy pomocy napisanej biblioteki. Testowany system posiada kilka cech, które wpłynęły na wybór narzędzia, a nie są to typowe cechy każdego systemu rozproszonego:

- znaczne wykorzystanie zarówno mocy obliczeniowej, jak i systemu wejścia–wyjścia (komunikacji sieciowej oraz urządzeń blokowych),
- zastosowanie wielowątkowości,
- równoległość wielu operacji umożliwiająca zwiększenie wydajności przy korzystaniu z wielu procesorów.

Wiele systemów klastrowych wykazuje podobne cechy i prawdopodobnie będzie mogło skorzystać z przygotowanej biblioteki. Dla systemów rozproszonych innego typu należałoby ponownie rozważyć argumenty przedstawione w tej pracy i lepszym rozwiązaniem mogłaby się okazać inna technika wprowadzania awarii. Nawet w takim przypadku analiza wielu możliwych

metod wprowadzania usterek przedstawiona w niniejszej pracy może posłużyć do napisania własnej biblioteki testów.

Dodatek A

Zawartość płyty CD

Na załączonej płycie CD znajdują się:

- welnicki_mgr.pdf — praca magisterska w formacie PDF,
- commBlockLib/ — biblioteka do wprowadzania awarii,
- testy/ — programy wykorzystane do przeprowadzenia testów wydajności,
- staf/ — wersje instalacyjne pakietów STAF oraz STAX.

Bibliografia

[1] H.-J. Hxer, M. Waitz, V. Sieh, *Advanced virtualization techniques for FAUmachine*, Linux-Kongress 2004, Erlangen, Germany, September 7–10, 2004, pages 1–12

[2] B. White et. al., *An integrated experimental environment for distributed systems and networks*, Proceedings of the Fifth Symposium on Operating Systems Design and Implementation, pages 255–270, December 2002

[3] B. Chun, *DART: Distributed Automated Regression Testing for Large-Scale Network Applications*, Proceedings of the 8th International Conference on Principles of Distributed Systems, 2004, http://www.theether.org/papers/dart.pdf

[4] L. Rizzo, *Dummynet: a simple approach to the evaluation of network protocols*, ACM Computer Communication Review, Jan. 1997

[5] *FAUmachine Project*, http://www3.informatik.uni-erlangen.de/Research/FAUmachine/

[6] M. Toren, *How to Avoid Writing Linux Kernel Modules, for fun and non-profit*, http://michael.toren.net/slides/lkm-alternatives/

[7] Postel, J. (ed.), *Internet Protocol — DARPA Internet Program Protocol Specification*, RFC 791, http://www.ietf.org/rfc/rfc0791.txt

[8] S. Bajaj et al., *Improving Simulation for Network Research*, Technical Report 99–702b, 1999

[9] *Opis łaty loop_discard w archiwach grupy dyskusyjnej linux-kernel*, http://www.kerneltraffic.org/kernel-traffic/kt20001016_89.html#1

[10] S. Dawson, F. Jahanian, T. Mitton, *ORCHESTRA: A Fault Injection Environment for Distributed Systems*, University of Michigan Technical Report CSE-TR-318-96, EECS Department

[11] *QEMU*, http://fabrice.bellard.free.fr/qemu/

[12] *Software Testing Automation Framework*, http://staf.sourceforge.net/

[13] *Strony podręcznika man GNU LD*(1)

[14] *Strony podręcznika man ld.so*(8)

[15] *Testdrive — łata na jądro Linuksa*, http://people.redhat.com/sct/patches/testdrive/

[16] V. Sieh, K. Buchacker, *Testing the Fault-Tolerance of Networked Systems*, 2002, 95–105, International Conference on Architecture of Computing Systems ARCS 2002, Workshop Proceedings

[17] *The Linux Test Project*, http://ltp.sourceforge.net/

[18] *The netfilter/iptables project*, http://www.netfilter.org/

[19] *The Network Simulator — ns-2*, http://www.isi.edu/nsnam/ns/

[20] C. Rankin, *The Software Testing Automation Framework*, http://www.research.ibm.com/journal/sj/411/rankin.html

[21] *The User-mode Linux Kernel*, http://user-mode-linux.sourceforge.net/

[22] *The Xen virtual machine monitor*, http://www.cl.cam.ac.uk/Research/SRG/netos/xen/

[23] *UML simulator*, http://umlsim.sourceforge.net/

[24] *UML simulator*, Ottawa Linux Symposium, July 2003

[25] *Universal TUN/TAP driver*, http://vtun.sourceforge.net/tun/

[26] *Xen and the Art of Virtualization*, SOSP 2003