Uniwersytet Łódzki
Wydział Matematyki

Marcin Gryszkalis

Rozproszony system udostępniania zasobów oparty na zdalnym wywoływaniu procedur

Praca wykonana
w Zakładzie Informatyki
Stosowanej pod kierunkiem
dr Mariusza Jarockiego

Łódź 2001

Spis treści

Wstęp 4

Rozdział 1. Rozproszone systemy plików 6

1.1. Cechy i założenia projektowe 6
1.1.1. Wprowadzenie 6
1.1.2. Nazewnictwo i jego przezroczystość 7
1.1.3. Usługi plikowe 8
1.1.4. Semantyka dzielenia plików 9
1.1.5. Serwery stanowe i bezstanowe 10
1.1.6. Podsumowanie wymagań projektowych 11
1.2. Implementacja 13
1.2.1. Wprowadzenie 13
1.2.2. Pamięć podręczna 14
1.2.3. Zwielokrotnianie 17
1.2.4. Wnioski 19
1.3. Zdalne wywołanie procedur 19
1.3.1. Wprowadzenie 19
1.3.2. Realizacja 20
1.3.3. Zagadnienia związane z przezroczystością 22
1.3.4. Przykładowe implementacje 24
1.4. Przykłady rozproszonych systemów plików 25
1.4.1. Unix United 25
1.4.2. Sun NFS 26
1.4.3. Andrew 27
1.4.4. DCE DFS 29
1.4.5. Amoeba 29
1.4.6. Sprite 30
1.4.7. Locus 31

Rozdział 2. rozproszony system plików DGFS 33

2.1. Projekt 33
2.1.1. Podstawy projektu 33
2.1.2. Nazewnictwo 34
2.1.3. Katalogi 34
2.1.4. Pliki; Pamięć podręczna i zwielokrotnianie 35
2.1.5. Semantyka współdzielenia plików 35
2.1.6. Awarie; Działanie w odłączeniu 36
2.1.7. Skalowalność i wydajność 36
2.2. Implementacja 36
2.2.1. Struktura systemu 37

2.2.2. Biblioteka programisty 37
2.2.3. Nadrzędny moduł uruchomieniowy 40
2.2.4. Moduł klienta 40
2.2.5. Moduł struktury drzewa 43
2.2.6. Moduł obsługi katalogów 45
2.2.7. Moduł obsługi plików 47
2.2.8. Moduł transportowy 49
2.2.9. Moduł dziennika 50
2.2.10. Moduł kontroli asynchronicznej 51
2.2.11. Przykładowe scenariusze zdarzeń w systemie 51
2.3. Możliwe rozszerzenia 53

Spis tabel 56

Spis rysunków 57

Bibliografia 58

Wstęp

System plików stanowi zasadniczy element każdego systemu komputerowego wywodzącego się z Unixa. Zwykle udostępnia on mechanizmy pozwalające operować na pamięciach masowych (tj. dyski, streamery) czyli na „typowych” plikach, ale także dostarczające dróg dostępu do urządzeń zewnętrznych (np. drukarki), pamięci operacyjnej czy struktury procesów wykonywanych na danym komputerze. Nie jest przesadnym stwierdzenie, iż w Unixie „wszystko jest plikiem”, gdyż praktycznie dowolny zasób może być wykorzystany za pomocą standardowych operacji *czytaj* i *pisz*.

Widać, że postać, efektywność działania i inne cechy systemu plików przenoszą się bezpośrednio na jakość systemu operacyjnego. Powodowane jest to między innymi przez przesunięcie profilu działania systemów z zastosowań „obliczeniowych” w kierunku przetwarzania ogromnych ilości danych.

Gwałtowny rozwój technologii sieciowych pokazał konieczność stworzenia technik udostępniania zasobów poprzez sieć. W początkowej fazie motywacją były ceny sprzętu — gdy nie można było sobie pozwolić na posiadanie twardego dysku w każdej stacji roboczej rozsądnym rozwiązaniem było współdzielenie jednego dysku między wiele komputerów. Dzisiaj, mimo iż argument cenowy stracił znaczenie, nadal projektuje się rozwiązania pozwalające współdzielić pliki. Powodem jest wygoda pracy, efektywność działania i inne cechy charakterystyczne dla systemów umożliwiających zdalny dostęp do zasobów (opisane zostaną szczegółowo w rozdziale pierwszym).

Wiele z technologii zdalnego dostępu do plików powstało razem z tworzonymi rozproszonymi systemami operacyjnym (podobnie jak „zwykły” lokalny system operacyjny opiera się na lokalnym systemie plików, tak rozproszony system operacyjny opiera się na rozproszonym systemie plików). Jedną z głównych cech, pożądanych w modelu rozproszonego systemu operacyjnego jak i w modelu jego systemu plików jest *nierozróżnialność* z odpowiednikiem lokalnym.

Rozdział pierwszy niniejszej pracy przedstawia różnorodne zagadnienia związane z projektowaniem rozproszonego systemu udostępniania zasobów, ich wady i zalety. Opisane zostaną technologie wykorzystywane w tworzeniu takich systemów (przede wszystkim *zdalne wywoływanie procedur*). W części końcowej zostaną zaprezentowane przykłady kilku konkretnych implementacji rozproszonych systemów plików.

Rozdział drugi przedstawia autorską, zaprojektowaną i stworzoną od podstaw implementację rozproszonego systemu plików DGFS (ang. *Distribut-*

ed Generic File System). Porównane zostaną założenia projektowe tego systemu z teoretycznymi rozwiązaniami prezentowanymi w literaturze a także z rozwiązaniami stosowanymi w innych rozproszonych systemach plików. Przedstawione i uzasadnione zostaną konkretne rozwiązania i sposoby wykorzystania systemu. Zaprezentowany zostanie także szczegółowy opis interfejsów modułu użytkownika i interfejsów wewnętrznych oraz wykorzystywanych struktur danych.

Rozdział 1

Rozproszone systemy plików

1.1. Cechy i założenia projektowe

Rozproszony system plików jest rozproszoną implementacją klasycznego modelu systemu plików, w którym wielu użytkowników dzieli pliki (i inne zasoby). Zadaniem rozproszonego systemu plików jest umożliwienie tego samego rodzaju dzielenia zasobów w warunkach fizycznego rozmieszczenia plików na różnych, połączonych siecią komputerach.

1.1.1. Wprowadzenie

Aby przejść do szczegółowego opisu cech rozproszonego systemu plików zdefiniowanych zostanie kilka podstawowych pojęć. Określenie *maszyna* używane będzie w odniesieniu do każdego komputera w sieci, niezależnie od jego logicznej roli. Określenia *zdalny* i *lokalny* w sposób naturalny odnoszą się do fizycznego umiejscowienia zasobu (plik jest lokalny dla tej maszyny, na której dysku fizycznie się on znajduje, dla pozostałych maszyn jest on zdalny). *Serwer* jest to oprogramowanie uruchomione na pewnej maszynie, które zajmuje się świadczeniem *Usług* (ang. *service*). *Klient* to proces, który zamawia usługę od serwera (zbiór usług dostępnych dla klientów nazywany jest *interfejsem klienta*.

Używając powyższej terminologii w odniesieniu do systemu plików można powiedzieć, że serwery (wchodzące w skład rozproszonego systemu plików) dostarczają swoim klientom usług plikowych (interfejs klienta zawiera elementarne usługi takie jak tworzenie pliku, kasowanie pliku, czytanie i zapis do pliku) oraz usług katalogowych (usługi tworzenia i kasowania katalogu, oraz odczytywania jego zawartości — nazw i atrybutów plików).

Idealny rozproszony system plików powinien wydawać się swoim klientom zwykłym, lokalnym systemem plików. Wielość i rozproszenie jego serwerów powinny być przezroczyste (ang. *transparent*). Oznacza to, że klient takiego systemu nie powinien odróżnić plików lokalnych od zdalnych (i do obydwu rodzajów powinien odwoływać się w jednolity sposób). Dodatkowym parametrem przezroczystości jest wydajność systemu plików (dąży się do tego aby wydajność rozproszonego systemu plików była porównywalna z wydajnością systemu lokalnego).

1.1.2. Nazewnictwo i jego przezroczystość

Usługi katalogowe

Interfejs systemu plików można podzielić na dwie zasadnicze części. Pierwsza obejmuje usługi plikowe (operacje na konkretnych plikach, tj. czytanie, pisanie), druga natomiast usługi katalogowe (tworzenie i usuwanie katalogów, nazywanie plików; zauważmy, że kasowanie pliku jest faktycznie operacją na katalogu i nie odnosi się w żaden sposób do zawartości pliku). Przedstawione zostaną teraz cechy interfejsu usług katalogowych w rozproszonym systemie plików.

Nazewnictwo jest to w ogólności odwzorowanie między obiektami logicznymi a fizycznymi. W lokalnym systemie plików możemy mieć do czynienia z odwzorowaniem nazwy pliku (obiekt logiczny, o którym myśli klient) na bloki dyskowe (na których operuje system operacyjny). Takie odwzorowanie ukrywa przed użytkownikiem szczegóły związane z techniczną stroną przechowywania danych na dysku.

W przezroczystym (ang. *transparent*) rozproszonym systemie plików dodawany jest kolejny poziom abstrakcji — ukrywa się przed klientem miejsce przechowywania pliku w sieci. Co więcej realizując taką przezroczystość można ukryć więcej niż tylko położenie pliku — na przykład dla użytkownika może być niewidoczne to, że plik znajduje się w sieci w wielu kopiach a klienci korzystają z dowolnej z nich (jest to tzw. zwielokrotnianie, opisane szerzej w rozdziale 1.2.3).

Struktury i schematy nazewnicze

Istnieją trzy podejścia do nazywania plików i katalogów (oraz odwzorowywania nazw w rozproszonym systemie plików):

jawność położenia: nazwa pliku (ścieżka) określa w sposób jawny gdzie znajduje się dany plik.

przezroczystość położenia: nazwa pliku nie zawiera informacji na temat fizycznego miejsca przechowywania pliku (ang. *location transparency*).

niezależność położenia: nazwy pliku nie trzeba zmieniać gdy plik zmienia swoje fizyczne umiejscowienie (ang. *location independency*).

Ścieżka w rodzaju `/maszyna1/kat1/kat2/plik1` mówi, że `plik1` znajduje się na serwerze `maszyna1`. Przemieszczenie pliku na inny serwer powoduje konieczność użycia innej ścieżki (np. `/maszyna2/kat1/kat2/plik1`). Jest to przykład systemu z częściową przezroczystością (częściową, gdyż ścieżka nie określa położenia samego serwera, lecz jest to cecha, którą raczej trudno wykorzystać praktycznie).

W systemach opartych na montażu[1] sytuacja wygląda podobnie — po przeniesieniu pliku na inny serwer nie można jednocześnie używać innych plików z `maszyny1` i z nowego serwera w ramach tego samego katalogu. Jednak dla użytkownika nieświadomego (tzn. takiego, który nie wie, że nastąpiło montowanie katalogów) system ten jest całkowicie przezroczysty pod względem położenia.

W praktyce niezależność położenia jest dużo silniejszą cechą systemu rozproszonego (trudniejszą do zrealizowania, jakkolwiek pożądaną). Większość rozproszonych systemów plików realizuje jedynie przezroczystość położenia. Widać, iż niezależność położenia jest równoznaczna z możliwością płynnego przemieszczania plików w systemie, co z kolei powoduje konieczność dynamicznego ich wyszukiwania (ponieważ między dwoma kolejnymi odwołaniami do pliku mógłby on zmienić swoje położenie, tj. zostać przeniesiony na inną maszynę). Wydaje się, że aby stworzyć rozproszony system plików funkcjonalnie w pełni odpowiadający systemowi lokalnemu, to konieczne jest uwzględnienie niezależności położenia.

1.1.3. Usługi plikowe

Żądanie dostępu do pliku realizowane jest w dwóch etapach. Pierwszy to lokalizacja pliku według podanej przez klienta nazwy za pomocą odpowiednich schematów nazewniczych. Drugi to realizacja rzeczywistego przesłania danych.

Modele dostępu

Usługi plikowe można podzielić na dwa rodzaje w zależności od tego, czy tworzą model pobierania-odsyłania (ang. *download-upload*), czy też model zdalnego dostępu (ang. *remote access*). Pierwszy model polega na tym, że cały plik, do którego klient żąda dostępu, zostaje sprowadzony przez sieć, a następnie oddany do dyspozycji żądającego. Wtedy klient może wykonywać na nim dowolne operacje (czytać, modyfikować) a po ich zakończeniu cały plik zostaje odesłany na maszynę zdalną (oczywiście odesłanie jest konieczne jedynie wtedy gdy klient zmodyfikował plik). Zaletą modelu pobierania-odsyłania jest prostota koncepcji. Podejście takie jest dobrze dopasowane do zauważonych rzeczywistych cech systemów plików (są one zebrane w tabeli 1.3 na stronie 13).

Drugi model (zdalnego dostępu) działa na zasadzie dokładnej realizacji żądań klienta — jeśli klient chce odczytać pierwszą linijkę pliku to właśnie ta linijka jest dla niego sprowadzana z maszyny zdalnej[2]. Zaletą tego modelu jest

[1] montażem określa się metodę dostępu do zdalnych zasobów poprzez „zamontowanie” (ang. *mount*) ich jako podkatalogu lokalnego systemu plików; w systemach unixowych służy do tego polecenie systemowe `mount`, przykładowo `mount /serwer1/kat /zdalny` może określić, że do zdalnego katalogu `kat` można się odwoływać poprzez lokalny katalog o nazwie `zdalny`

[2] w rzeczywistości zazwyczaj jednostką transferu są bloki o pewnej ustalonej stałej długości, np. 4KB, gdyż pobieranie pojedynczych bajtów byłoby bardzo nieefektywne

mniejsze zapotrzebowanie na zasoby lokalne klienta (może to być maszyna nie posiadająca dysku i z ograniczoną pamięcią operacyjną, w przypadku modelu sprowadzania-odsyłania potrzebna jest pamięć zdolna pomieścić cały używany plik).

1.1.4. Semantyka dzielenia plików

Gdy dwu lub więcej użytkowników dzieli ten sam plik, wówczas w celu uniknięcia problemów konieczne staje się zdefiniowanie semantyki operacji odczytu i zapisu. W systemach lokalnych takich jak Unix, semantyka taka stanowi, że jeśli operacja *czytaj* nastąpiła po operacji *pisz*, to *czytaj* zwraca wartość dopiero co zapisaną. Natomiast jeżeli bezpośrednio po sobie następują dwie operacje *pisz* to kolejna operacja *czytaj* zwraca wartość zapisaną przez późniejszą operację *pisz*. Ten prosty model nazywany jest semantyką unixową.

Zachowanie pełnej semantyki unixowej nie jest w systemie rozproszonym możliwe ze względu na niemożność pełnej synchronizacji czasu[3] poprzez sieć (tzn. jeśli dwóch klientów A i B wykonało operację *pisz* na pewnym pliku, przy czym A wykonał ją najpierw, to informacje o tych operacjach mogą trafić do serwera w dowolnej kolejności — zależy to m.in. od opóźnień sieci). Często należy brać pod uwagę, że dany plik może być przechowywany w pamięci podręcznej lub być dostępny w postaci repliki co dodatkowo utrudnia zachowanie semantyki (zagadnienia replikacji i spójności pamięci podręcznej zostaną omówione w dalszej części).

Warto również zauważyć, że operacje *pisz* i *czytaj* są zazwyczaj operacjami jądra systemu, a w przypadku rozproszonych systemów operacyjnych, samo jądro jest rozproszone i co za tym idzie operacje wykonywane w takim jądrze nie muszą być uszeregowane (w zwykłym jądrze lokalnego systemu zawsze można określić relację następstwa czasu).

Alternatywnym rozwiązaniem dla semantyki unixowej jest tzw. semantyka sesji, która działa w ten sposób, że zmiany dokonane w pliku są widoczne dla innych klientów dopiero po zamknięciu pliku (pojedyncze modyfikacje są widoczne jedynie dla tego, kto je wykonuje). Semantyka sesji jest najpopularniejszym rozwiązaniem w systemach rozproszonych.

Trzecie podejście polega na uczynieniu wszystkich plików stałymi. Znaczy to, że niemożliwa jest faktyczna modyfikacja pliku gdyż jedynymi dopuszczalnymi operacjami są *twórz plik* i *czytaj*. Nie występuje tu sytuacja gdy jeden klient czyta a drugi pisze do pliku (gdyż nie ma operacji *pisz*) — natomiast może wystąpić sytuacja, w której jeden klient czyta plik a drugi usiłuje utworzyć nowy pod tą samą nazwą (co jest z założenia możliwe). Wtedy zazwyczaj pozwala się pierwszemu klientowi dokończyć czytanie ze starego pliku (mimo iż nie jest już dostępny pod swoją nazwą), natomiast

[3] zagadnienie synchronizacji czasu w systemach rozproszonych jest bardzo trudnym, dokładnie badanym i szeroko opisanym w literaturze problemem

wszystkie następne otwarcia będą się odnosić do nowego, utworzonego przez drugiego klienta pliku.

Czwarty sposób postępowania polega na użyciu niepodzielnych transakcji (działających na podobnej zasadzie co transakcje w bazach danych). Klient chcący wykonać pewne operacje *otwiera* transakcję i do czasu jej *zamknięcia* nikt inny nie może dokonać zmian w obiektach, do których ten klient się odwołuje. Mówimy, że pliki objęte transakcją są zablokowane lub inaczej zajęte (ang. *locked*).

W tabeli 1.1 zebrane są cztery omówione sposoby postępowania z plikami dzielonymi w systemach rozproszonych.

Metoda	Komentarz
Semantyka unixowa	Każda operacja na pliku jest natychmiast widoczna dla wszystkich procesów.
Semantyka sesji	Żadne zmiany nie są widoczne dla innych procesów do czasu zamknięcia pliku.
Pliki stałe	Niedozwolone są żadne aktualizacje, co upraszcza dzielenie i zwielokrotnianie.
Transakcje	Zmiany odbywają się na zasadzie „wszystko albo nic". Zasoby są blokowane.

Tabela 1.1: Cztery sposoby postępowania z plikami dzielonymi w systemie rozproszonym

1.1.5. Serwery stanowe i bezstanowe

Istnieją dwa podejścia do informacji przechowywanej po stronie serwera. Serwer może po prostu przesyłać bloki do i od klienta lub też pamiętać pomiędzy kolejnymi żądaniami jego stan.

Pierwszy wariant to tak zwany model serwera bezstanowego (ang. *stateless*). Serwer realizuje zamówienie, odpowiada na żądanie klienta i usuwa ze swoich struktur wewnętrznych informacje o zrealizowanym zamówieniu. Pomiędzy kolejnymi zamówieniami nie przetrzymuje żadnych informacji o kliencie (następstwem tego jest jego niewiedza na temat czy dwa kolejne zamówienia były wysłane przez tego samego klienta czy przez dwóch różnych). W interfejsie serwera nie ma takich operacji jak *otwórz plik* i *zamknij plik* — każde zamówienie musi dokładnie określić, do którego pliku się odnosi i gdzie znajdują się dane, których ono dotyczy.

Drugi wariant to model serwera stanowego (ang. *stateful*). Jest to przeniesienie tradycyjnych lokalnych metod dostępu do plików na system rozproszony. Tutaj serwer po otwarciu pliku przez klienta zwraca mu pewien unikalny identyfikator, który będzie następnie wykorzystywany w kolejnych operacjach (np. *czytaj*, *pisz* i *zamknij plik*). Serwer musi przechowywać tabli-

cę, w której kojarzy ten identyfikator z nazwą pliku, trybem jego otwarcia, aktualną pozycją w tym pliku i innymi potrzebnymi informacjami.

Znaczącą zaletą serwerów bezstanowych jest dużo lepsze tolerowanie awarii — zarówno awarii serwera jak i klientów. Jeżeli serwer stanowy w wyniku problemów straci informacje o otwartych plikach to klienci nie mogą dokończyć rozpoczętych działań, muszą rozpoczynać od nowa a czasem odtworzenie sytuacji poprawnej może być bardzo trudne. W przypadku serwera bezstanowego klient po prostu ponawia żądanie aż do skutku (jakkolwiek należy pamiętać wtedy, musi być zapewniona idempotentność[4] zamówień. W przypadku awarii klienta serwer pozostaje z otwartym plikiem, do którego nikt się już nie odwołuje — nie może określić czy brak zamówień wynika z tego, że klient jest wolny czy może z tego, że wystąpił problem przy transmisji siecią.

Zaletą serwerów stanowych jest możliwość realizacji semantyki sesji lub semantyki transakcji (obie semantyki zakładają przechowywanie informacji o stanie klienta). Porównanie serwerów bezstanowych i stanowych zawiera tabela 1.2.

Podsumowując — wybór pomiędzy wariantem stanowym a bezstanowym jest jednym z najważniejszych problemów, przed którymi staje projektant rozproszonego systemu plików. Decyzja oparcia systemu na jednym z tych modeli implikuje sposób realizacji bardzo wielu zachowań serwera. Wybór musi być podejmowany w oparciu o rzeczywiste (lub przewidywane) warunki w jakich będzie pracował system — na przykład w środowisku silnie narażonym na awarie uzasadnione będzie użycie serwera bezstanowego, natomiast gdy priorytetem jest skuteczna i bezpieczna obsługa wielodostępności (dzielenia) plików konieczne będzie zastosowanie systemu stanowego umożliwiającego wykorzystanie blokowania plików (np. w semantyce transakcji).

1.1.6. Podsumowanie wymagań projektowych

Łącząc opisywane wyżej zagadnienia można stworzyć kompletną listę wymagań, które powinien spełniać dobry rozproszony system plików. Nie znaczy to, że wszystkie poniższe wymagania muszą być spełnione aby system działał poprawnie i efektywnie, co więcej, niektóre z nich stoją ze sobą w pewnej sprzeczności. Należy zachować równowagę pomiędzy przezroczystością i skalowalnością systemu a jego złożonością i wydajnością.

Główne wymagania projektowe rozproszonego systemu plików:

przezroczystość dostępu: Klienci nie są świadomi rozproszenia plików, dostęp do plików lokalnych i zdalnych odbywa się za pomocą jednego zbioru operacji.

przezroczystość i/lub niezależność położenia: Klienci mają do czynienia z jednolitą przestrzenią nazw plików. Po uwzględnieniu niezależności

[4] opis czym jest idempotentność oraz rozróżnienie między zamówieniami powtarzalnymi i niepowtarzalnymi znajduje się w rozdziale 1.3.3 dotyczącym awarii przy zdalnym wywoływaniu procedur

cecha	serwer bezstanowy	serwer stanowy
Tolerowanie uszkodzeń	Dobre i łatwe w realizacji	Trudne, wymagające specjalnych rozwiązań
Efektywność żądań	Każde żądanie musi zawierać komplet informacji	Krótkie komunikaty z zamówieniami
Zasoby	—	Potrzebna jest pamięć na przechowywanie tablic ze stanami klientów
Efektywność serwera	Dla każdego żądania serwer musi wykonać lokalnie dużo operacji (otwarcie pliku, itp.) co obniża efektywność	Działanie polega na prostym przeniesieniu żądania klienta do systemu plików serwera, dodatkowo serwer ma możliwość wykonania czytania z wyprzedzeniem
Ograniczenia	Nie można stosować niektórych semantyk dostępu do plików	Może wystąpić ograniczenie ze względu na liczbę otwartych plików

Tabela 1.2: Porównanie wad i zalet serwerów bezstanowych i stanowych

położenia pliki mogą zmieniać fizyczne położenie nie powodując zmian w nazwach.

przezroczystość współbieżności: Zmiany w pliku dokonane przez jednego klienta nie zaburzają działań innych klientów. Zapewnione jest stosowanie odpowiedniej semantyki dostępu do plików dzielonych.

przezroczystość awarii: Zapewnione jest poprawne działanie serwerów po awarii klienta oraz poprawne działanie klientów pomimo zagubienia informacji w sieci i przerw w działaniu serwerów.

przezroczystość wydajności: Programy klientów powinny być wykonywane w czasie „zadowalającym”, pomimo określonych opóźnień wynikających z działania systemu.

heterogeniczność: Chodzi tu o heterogeniczność sprzętu i systemu operacyjnego. Interfejsy powinny być tak zdefiniowane aby umożliwić ich wykorzystywanie w różnych systemach operacyjnych i na różnych komputerach (tak aby różne maszyny współpracowały w ramach jednego rozproszonego systemu plików).

skalowalność: Przy wzrastającym obciążeniu i rozmiarze sieci usługi powinny zachowywać zakładaną wydajność. Wzrost obciążenia oznacza zarówno zwiększenie ilości klientów w całym systemie jak i zwiększenie ilości żądań przypadających na jeden serwer. Inną miarą obciążenia jest wielkość danych przechowywanych w systemie — także tego czynnika dotyczy tutaj pojęcie skalowalności.

przezroczystość replikacji: Jeśli system implementuje replikację (zwielo-

krotnienie, patrz rozdział 1.2.3) to powinna być ona także przezroczysta dla klienta.

środki drobnoziarnistego rozproszenia danych: W systemach rozproszonych budowanych w oparciu o specyficzne założenia może być konieczne stworzenie mechanizmów pozwalających na większe rozproszenie informacji (np. fragmentację plików pomiędzy serwerami).

tolerowanie podziału sieci i działanie po odłączeniu: Przerwanie połączeń w sieci może być wynikiem awarii lub działań przemyślanych (np. odłączenie przenośnego komputera od sieci) — odpowiednio zaprojektowane mechanizmy pamięci podręcznej i replikacji mogą pozwolić na ograniczone lecz poprawne działanie klientów.

1.2. Implementacja

1.2.1. Wprowadzenie

Tworząc rozproszony system plików rozpoczyna się od wyboru i ustalenia odpowiednich założeń projektowych a następnie określa sposób ich realizacji. W planowaniu można posłużyć się opracowaniami dotyczącymi sposobów użytkowania plików w różnych systemach wybierając takie opracowanie, które najbliżej odpowiada środowisku, w którym ma działać opracowywany system.

Omówionych zostanie teraz kilka ogólnych obserwacji dotyczących systemów plików (zarówno lokalnych jak i rozproszonych) — odnoszą się one do większości typowych zastosowań i odzwierciedlają sposób działania standardowego wieloużytkownikowego systemu operacyjnego.

Najistotniejsze cechy systemów plików zebrane są w tabeli 1.3.

system plików
Pliki są w większości małe.
Czytanie występuje znacznie częściej niż pisanie.
Czytanie i pisanie odbywa się sekwencyjnie, dostęp swobodny występuje rzadko.
Wiele plików istnieje krótko.
Dzielenie plików (przy zapisie) występuje sporadycznie, zapisu do danego pliku dokonuje zazwyczaj tylko jeden konkretny klient.
Przeciętny proces używa tylko kilku plików.
Odwołania do pliku są skumulowane.
Istnieją różne klasy plików o różnych cechach.

Tabela 1.3: Obserwowane właściwości systemu plików

Warto w tym miejscu zauważyć, że większość założeń z poprzedniego rozdziału jak i obserwacji opisanych w tabeli 1.3 nie dotyczy systemów w których używane są bazy danych. Pomimo iż bazy danych faktycznie też mogą być dostępne poprzez, system plików to jednak wymagania jak i zachowania ich użytkowników są znacząco różne od wymagań i zachowań użytkowników zwykłego systemu plików (na przykład dla baz danych typowym sposobem dostępu jest dostęp swobodny a nie sekwencyjny). Oznacza to, że tworzenie rozproszonych baz danych musi być rozpatrywane jako zupełnie odrębny problem.

Z przedstawionych obserwacji wynikają następujące wnioski. Jeżeli większość plików ma stosunkowo mały rozmiar to można założyć, że przenoszenie pomiędzy serwerem a klientem całych plików będzie bardziej efektywne, niż sztuczne dzielenie na bloki. Oczywiście postęp technik multimedialnych sprawia, że coraz częściej mamy też do czynienia z dużymi, wielomegabajtowymi plikami — pokrywa się to z obserwacją, że istnieje kilka klas plików, które mogą wymagać różnego potraktowania.

Ciekawa jest obserwacja, iż wiele plików istnieje krótko (chodzi tu np. o pliki tymczasowe tworzone przez różne programy). Dodatkowo pliki tymczasowe nie są zazwyczaj przeznaczone do współdzielenia między programami na różnych maszynach. Automatycznie nasuwa się potraktowanie takich plików w specjalny sposób — jako plików „prywatnych”, tzn. nie informowanie o nich sytemu, co znacząco obniża ruch w sieci i obciążenie serwerów.

Fakt, że plików dzielonych jest mało, przemawia za szerokim zastosowaniem pamięci podręcznych i replikacji (co komplikuje utrzymanie odpowiedniej semantyki ale znacząco zwiększa wydajność).

Informacja, że zapisu do pliku dokonuje zazwyczaj ten sam klient sugeruje stworzenie mechanizmów ułatwiających pracę temu właśnie klientowi (rozwiązanie takie zostanie przedstawione w projekcie systemu DGFS w rozdziale drugim).

1.2.2. Pamięć podręczna

Schematy przechowywania podręcznego

Aby zapewnić zadowalające działanie systemu plików stosuje się różne formy przechowywania podręcznego lub inaczej pamięci podręcznej (ang. *cache*). W zwykłym systemie plików dąży się do zmniejszenia liczby czasochłonnych dyskowych operacji wejścia-wyjścia. W rozproszonych systemach plików zadaniem pamięci podręcznej jest zarówno zmniejszenie ilości operacji dyskowych jak i zmniejszenie ruchu w sieci (to drugie ma zazwyczaj większe znaczenie, gdyż wprowadza znacznie poważniejsze opóźnienia).

Jeśli chodzi o pamięć podręczną serwera obsługującą jego operacje dyskowe sytuacja jest dosyć jasna — efekty działania tej pamięci są zupełnie przezroczyste dla klientów, można zastosować właściwie dowolną technikę obsługi i uaktualniania, nie występuje tu także problem spójności pamięci podręcznej. Co więcej tego typu przechowywanie podręczne realizuje zazwy-

czaj system operacyjny serwera więc przy tworzeniu rozproszonego systemu plików można ten aspekt pominąć (chyba, że ten budowany system plików jest częścią całego, tworzonego od podstaw rozproszonego systemu operacyjnego). Widać także, że tego typu pamięć nie wpływa w żaden sposób na ograniczenie ilości danych przesyłanych siecią.

Dużo większy wpływ na wydajność może mieć pamięć podręczna umieszczona po stronie klienta. Jej działanie w przypadku ogólnym sprowadza się do tego, że jeżeli klient w pewnym odstępie czasu dwukrotnie odwołuje się do tego samego pliku, to za drugim razem zawartość pliku nie jest transportowana siecią lecz pobierana z pamięci podręcznej (może to być pamięć operacyjna klienta lub lokalny dysk klienta). Widać od razu, że pojawia się tu problem spójności pamięci podręcznej — przecież pomiędzy pierwszym a drugim odwołaniem do pliku jego zawartość na serwerze mogła ulec zmianie i wtedy klient otrzyma nieaktualne dane. Problem ten zostanie omówiony w dalszej części.

Budowa pamięci podręcznej

Niezależnie czy dotyczy to pamięci podręcznej umiejscowionej na dysku czy w pamięci operacyjnej, należy rozważyć kilka założeń, które mają wpływ na sposób implementacji odpowiednich mechanizmów jak i na wydajność. Jednym z nich jest ziarnistość pamięci podręcznej — można zapamiętywać małe bloki (odpowiadające wielkością blokom, w których odbywa się transmisja danych w sieci), duże bloki (których wielkość jest pewną, niekoniecznie stałą, wielokrotnością wielkości bloków transmisyjnych) lub całe pliki. Parametr ten można ustalić w oparciu o inne założenia projektowe.

Warto również zauważyć, że nie można budować pamięci podręcznej opartej na pamięci operacyjnej w oderwaniu od systemu operacyjnego — łatwo można wyobrazić sobie taką sytuację gdy zarządca pamięci znajdujący się w jądrze systemu operacyjnego zadecydował, że stronę pamięci zawierającą przechowywany podręcznie plik należy przenieść na dysk[5]. Wtedy odwołanie do tego pliku spowoduje błąd strony w systemie operacyjnym i dodatkowy zestaw operacji dyskowych w celu ponownego sprowadzenia danej strony do pamięci operacyjnej. Zaburza to całkowicie sens działania pamięci podręcznej.

Polityka uaktualniania

Polityka zastosowana przy odsyłaniu zmienionych bloków danych z powrotem do głównej kopii pliku na serwerze ma decydujące znaczenie dla działania i niezawodności systemu. Najprostsza polityka polega na przepisywaniu danych zaraz po tym, gdy znajdą się w pamięci podręcznej. Takie rozwiązanie nazywane jest algorytmem przepisywania (ang. *write-through algorithm*). Wadą tej polityki jest słaba wydajność operacji pisania, gdyż po każdym zapisie klient musi czekać na przesłanie danych do serwera. Oznacza to, że

[5] zagadnienia związane z zarządzaniem pamięcią operacyjną i stronicowaniem zostały szeroko opisane w [5]

efektywność pamięci podręcznej jest wykorzystywana jedynie przy czytaniu danych. Rozwiązanie takie zastosowano w systemie NFS (rozdział 1.4.2).

Alternatywna polityka polega na opóźnieniu aktualizacji kopii głównej. Zmiany są wprowadzane do pamięci podręcznej, a do serwera są przepisywane asynchronicznie z pewnym opóźnieniem (odpowiadającym przepustowości sieci i obciążeniu serwera). Oczywiście wydajność operacji pisania jest tutaj znacznie większa. Co więcej jeżeli klient kilkakrotnie aktualizuje te same dane w małych odstępach czasu to są one przepisywane do serwera tylko raz. Podstawową wadą tego rozwiązania jest niska niezawodność (w przypadku awarii dane, które (jak uważa klient) zostały zapisane tak, naprawdę nie trafiły do serwera. Polityka ta nazywana jest algorytmem opóźnionego pisania (ang. *delayed-write*).

Trzecie podejście jest nieco uproszczoną formą drugiego — polega ono na tym aby dane do serwera przesyłać po zamknięciu pliku przez klienta (ang. *write-on-close*). Jest to nieco łatwiejsze w realizacji gdyż nie wymaga działań asynchronicznych. Taka forma aktualizacji odpowiada semantyce sesji.

Spójność pamięci podręcznej

Zapewnienie zgodności kopii pliku przechowywanych podręcznie z plikiem głównym[6] (pierwotnym, umieszczonym na serwerze) nosi nazwę problemu spójności pamięci podręcznej (ang. *cache consistency problem*). Istnieją dwie metody rozwiązywania tego problemu:

sprawdzanie inicjowane przez klienta: Klient kontaktuje się z serwerem i sprawdza czy kopia lokalna jest zgodna z plikiem głównym. Może to być dokonane za pomocą odpowiednich znaczników — sprawdzanie odbywa się następująco: serwer za każdym razem gdy otrzymuje nową wersję pliku od jednego z klientów zwiększa pewien odpowiadający temu plikowi licznik. Licznik ten jest przekazywany razem z plikiem wszystkim klientom, którzy pobierają plik. Następnie jeżeli klient chce skorzystać z pliku znajdującego się w pamięci podręcznej porównuje jego licznik z licznikiem na serwerze — niezgodność liczników oznacza niespójność danych i wymusza ponowne pobranie pliku z serwera. W zależności od zastosowanej semantyki i sposobu dostępu sprawdzenie to może odbywać się przy każdym odwołaniu do pliku lub też tylko przy jego otwarciu. Wadą tej metody jest dodatkowe obciążenie sieci (które jest tym mniej potrzebne im rzadziej zmieniane są pliki).

uaktualnianie inicjowane przez serwer: Serwer rejestruje, dla każdego klienta z osobna, jakie pliki zostały pobrane do pamięci podręcznych i w momencie zmiany zawartości pliku głównego informuje wszystkich zainteresowanych klientów, że ich dane są już nieaktualne. Wadą tego rozwiązania jest to, że zamieniają się role klienta i serwera (a więc klient musi zacząć nasłuchiwać informacji ze strony serwera, co może skomplikować

[6] mówiąc ogólniej chodzi o zgodność pamięci podręcznej z najnowszą wersją pliku — istnieją systemy bez wyróżnionej kopii głównej (przykładem może być system DGFS)

jego budowę). Poza tym może się zdarzyć, że komunikat od serwera zaginie i klient nie będzie świadomy, że posiada nieaktualne dane[7]. Oczywiście zaletą tego rozwiązania jest znikomy ruch w sieci w przypadku gdy pliki są rzadko zmieniane.

1.2.3. Zwielokrotnianie

Powielanie plików na różnych maszynach powoduje użyteczną nadmiarowość poprawiającą dostępność usług plikowych. Zwielokrotnianie (lub inaczej replikacja) informacji ma korzystny wpływ na wydajność, gdyż wybór najbliższej kopii skraca czas obsługi zamówienia a obciążenie rozkłada się pomiędzy wiele serwerów i wiele segmentów sieci.

Zastosowania replikacji

Można mówić o trzech głównych powodach, dla których stosuje się zwielokrotnianie:

1. Poprawa wydajności działania rozproszonego systemu plików jako całości przez równomierne rozłożenie obciążenia pomiędzy serwery oraz przez skrócenie wędrówek danych po sieci.
2. Umożliwienie dostępu do pliku nawet w przypadku niedostępności jednego z serwerów.
3. Zwiększenie niezawodności przez utrzymywanie niezależnych kopii pliku. W przypadku awarii nie giną żadne dane oraz jest możliwość szybkiego odtworzenia normalnego stanu (zazwyczaj za pomocą samych mechanizmów zwielokrotniania).

System plików zasadniczo powinien być przezroczysty względem zwielokrotniania — co znaczy, że struktury nazewnicze nie powinny dawać użytkownikowi informacji co do tego, która z replik jest używana. Co więcej pozwolenie użytkownikowi na jawne określenie, której kopii chce on używać może prowadzić do zmniejszenia efektywności działania systemu — często system „lepiej wie” co jest dla klienta dobre, może na przykład badać transfery w sieci i na tej podstawie wybierać repliki. Wydaje się, że nie jest trudne stworzenie tak działających mechanizmów. Oczywiście należy zostawić sobie możliwość sterowania zwielokrotnianiem w sensie ustalania ogólnych zasad, według których ma się ono odbywać.

Metody zwielokrotniania

Istnieją trzy podstawowe techniki wykonywania replikacji:

zwielokrotnianie ręczne i jawne: Osoba (może to być też program) nadzorująca pracę systemu posługując się jakimiś kryteriami tworzy kopie plików na wybranych serwerach informując o tym część systemu odpowiedzialną za wyszukiwanie plików (usługi nazewnicze).

[7] problem niezawodności transmisji jako takiej rozwiązywany jest zazwyczaj na innym poziomie — na przykład na poziomie protokołów TCP/IP

leniwe zwielokrotnianie: W trakcie działania systemu tworzone są automatycznie kolejne kopie — odbywa się to na zasadzie rozchodzenia się plotki[8]. Zwielokrotnianie takie wykorzystuje czas, w którym serwer nie jest wykorzystywany.

zwielokrotnianie grupowe: Ten model wykorzystuje komunikację grupową[9]. Polega ona na wysyłaniu jednego komunikatu, który jest odbierany przez wszystkie maszyny znajdujące się w pewnej logicznej grupie. Pozwala to na szybkie wykonanie replikacji przy mniejszym całkowitym ruchu w sieci, lecz niesie ze sobą pewne problemy związane z samą technologią komunikacji grupowej.

Metody aktualizacji

Istnieje także kilka algorytmów organizujących aktualizację replik, przy czym zazwyczaj jest tak, że rozwiązania posiadające ciekawe cechy (na przykład odporność na błędne działanie serwerów) stawiają nierealne wymagania (takie jak konieczność kontaktu z ponad połową serwerów w systemie przy każdym odczycie z pliku). Dwa najważniejsze algorytmy to:

zwielokrotnianie kopii podstawowej: Przy zastosowaniu tego algorytmu jeden z serwerów pełni rolę podstawowego (wszystkie inne są wtórne). Gdy pojawia się konieczność uaktualnienia zwielokrotnionego pliku, informacje o tym, co ma być zmienione wysyła się do serwera podstawowego, który dokonuje zmian swojej kopii, po czym wysyła informacje o zmianach do serwerów wtórnych. Metoda ta ma jedną wadę — w przypadku awarii lub braku dostępu do serwera głównego nie jest możliwe wykonanie żadnej aktualizacji. Problem ten próbują rozwiązać inne algorytmy.

głosowanie: Jest to dość skomplikowany algorytm polegający na tym, że klient chcący zmienić plik kontaktuje się z ponad połową serwerów posiadających kopie danego pliku. Wszystkie te serwery określają numery wersji pliku jakie posiadają i odpowiednio je aktualizują. Także przy czytaniu klient musi się kontaktować z ponad połową serwerów — zgodnie z tym w jaki sposób dokonywana jest aktualizacja, wśród skontaktowanych serwerów musi być co najmniej jeden posiadający najnowszą wersję pliku. Metoda ta, mimo iż w rzeczywistości realizuje podstawowe pragnienie projektantów systemów rozproszonych (brak wyróżnionego koordynatora), sprawia wiele problemów (szczególnie konieczność kontaktowania się z dużą liczbą serwerów przy czytaniu pliku jest tutaj czynnikiem mocno obciążającym wydajność).

---

[8] określenie plotka (ang. *gossip*) jest rzeczywistym określeniem używanym w literaturze; można spotkać się też z „architekturą plotki”

[9] nie wszystkie systemy mają zaimplementowany ten rodzaj komunikacji; niezawodność komunikacji grupowej jest jednym z podstawowych problemów badanych przez projektantów systemów rozproszonych

1.2.4. Wnioski

Opierając się na doświadczeniach istniejących i tworzonych systemów plików można sformułować kilka podstawowych zasad implementacji:

- Pamięć podręczna, gdzie tylko możliwe — jest to dość oczywista zasada.
- Wykorzystywanie cech użytkowych — należy przy implementacji brać pod uwagę rzeczywiste cechy środowiska w jaki będzie pracował system (część takich cech została omówiona w tabeli 1.3 na stronie 13).
- Minimalizowanie informacji ogólnosystemowych a także ich zmian — jak największa ilość informacji powinna być przechowywana lokalnie, np. nie należy żądać aby każdy serwer miał informacje o całym drzewie katalogów, a tylko o pewnej jego części. W realizacji tej zasady pomaga wykorzystanie struktur hierarchicznych (właśnie takich jak drzewo katalogów).
- Ufanie najmniejszym możliwym całościom — jeśli poprawne funkcjonowanie systemu miałoby zależeć od właściwego działania wszystkich stacji roboczych to szansa na dłuższą pracę byłaby znikoma.
- Kumulowanie pracy — oznacza to łączenie drobnych elementów w większe całości (np. przy transmisji danych czy komunikatów)

1.3. Zdalne wywołanie procedur

1.3.1. Wprowadzenie

W trakcie projektowania i tworzenia rozproszonego systemu operacyjnego jednym z największych problemów jest zapewnienie odpowiedniej komunikacji pomiędzy węzłami sieci — jednostkami wchodzącymi w skład systemu. Podstawowym modelem, na jakim oparta jest komunikacja jest model klient-serwer. Komunikacja w tym modelu zbudowana jest zwykle w oparciu o ciągi wywołań typu *wyślij żądanie* i *odbierz odpowiedź* — co powoduje, iż logika działania systemu jest przysłonięta problemami obsługi wejścia-wyjścia. Widać, że jest to sprzeczne z ideą uczynienia przetwarzania rozproszonego podobnym do przetwarzania przeprowadzanego lokalnie, zatem celem projektanta systemu rozproszonego jest tutaj możliwie pełne ukrycie infrastruktury komunikacyjnej.

Jednym ze sposobów osiągnięcia przezroczystości rozproszenia jest zdalne wywołanie procedur — w skrócie RPC (ang. *Remote Procedure Call*). Istotą tego rozwiązania jest „niewidoczna” dla piszącego program podmiana procedury lokalnej na procedurę zdalną. Gdy program wykonywany na komputerze A wywołuje funkcję f() jego działanie zostaje zawieszone i na maszynie B (określonej w sposób jawny lub nie) następuje wykonanie funkcji f(). Po zakończeniu wykonania proces na maszynie A otrzymuje wyniki działania tejże funkcji. Programista nie ma do czynienia z żadnym przekazywaniem komunikatów ani operacjami wejścia-wyjścia.

Jakkolwiek idea jest prosta i efektowna, wymaga jednak rozwiązania kilku problemów. Po pierwsze — skoro program wywołujący procedurę i sama

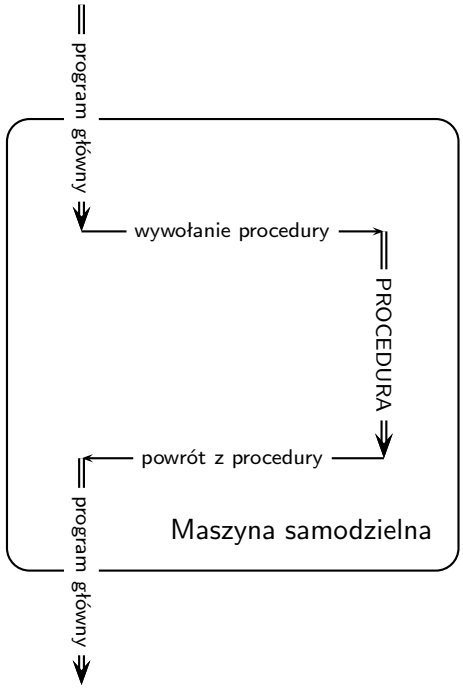
procedura znajdują się na różnych komputerach — należy przesyłać argumenty i wyniki działania funkcji korzystając z zawodnej warstwy sieciowej. Drugim problemem jest heterogeniczność maszyn (głównie jeśli chodzi o typy i reprezentacje danych — oczywiste jest, że jeżeli funkcja została wywołana z parametrem o wartości 1, to z takim samym parametrem musi być wykonana).

W chwili obecnej RPC jest jedną z podstawowych technik, na której bazują systemy rozproszone. Istnieje wiele standardowych usług w systemach unixowych korzystających z RPC (na przykład zaprojektowany przez firmę Sun rozproszony system plików NFS — *Network File System*, opisany w rozdziale 1.4.2).

Powstały i rozwinęły się też systemy rozszerzające funkcjonalność klasycznego RPC (m.in. o obiektowość) — np. system CORBA (ang. *Common Object Broker Request Architecture*) czy zaprojektowany przez firmę Microsoft *Distributed COM*.

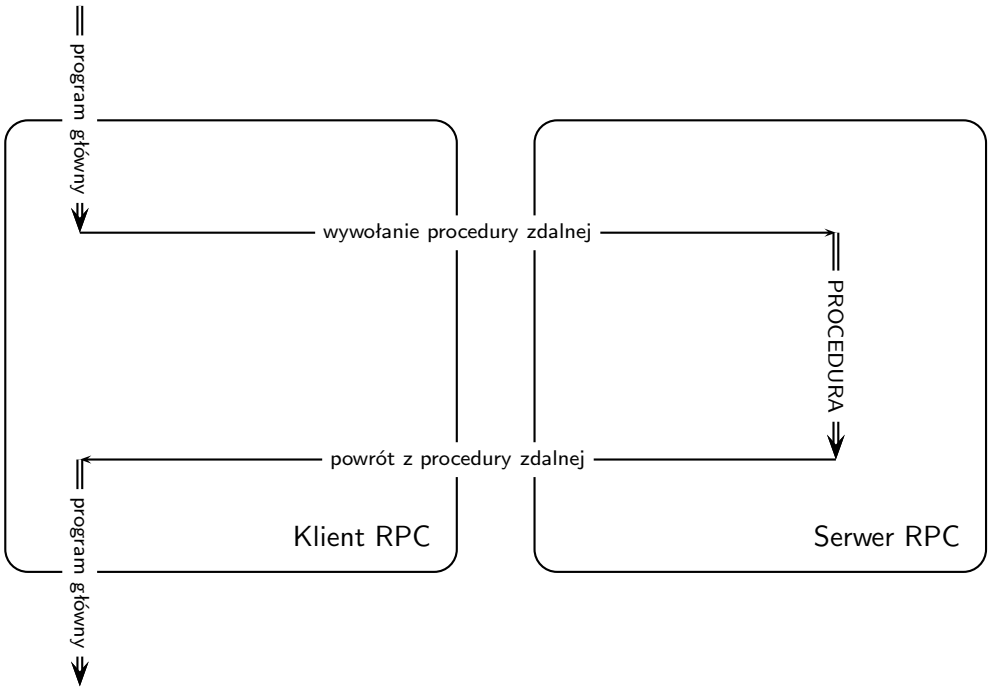
1.3.2. Realizacja

Porównane zostaną teraz szczegóły modeli wywołania lokalnego i zdalnego (oba modele są także przedstawione na rysunkach 1.1 i 1.2).



Rysunek 1.1: Schemat lokalnego wywołania procedury

Zwykłe (lokalne) wywołanie procedury można opisać następującymi krokami:



Rysunek 1.2: Schemat zdalnego wywołania procedury

1. program odkłada na stos argumenty procedury
2. następuje skok do procedury lokalnej, która zdejmuje argumenty ze stosu i wykonuje swoje zadanie, następnie odkłada wyniki na stos i wykonuje instrukcję powrotu z procedury
3. program zdejmuje wyniki działania procedury ze stosu

Natomiast zdalne wywołanie procedury przebiega w sposób następujący:

1. program odkłada na stos argumenty procedury
2. następuje skok do lokalnej procedury zwanej pieńkiem klienta(ang. *client stub*)[10] — jest to substytut rzeczywistej procedury, którego zadaniem przeprowadzenie komunikacji z serwerem, na którym owa rzeczywista procedura ma być wykonana; pieniek zdejmuje ze stosu parametry a następnie formuje je (ang. *parameters marshalling*) tak aby mogły zostać przesłane przez sieć
3. komunikat z parametrami zostaje przesłany do pieńka serwera (ang. *server stub*), który dokonuje rozformowania (ang. *unmarshalling*) parametrów i odkłada je na stos (ważne, że jest to stos serwera)
4. następuje skok do procedury lokalnej (serwera), która zdejmuje argumenty ze stosu i wykonuje swoje zadanie, następnie odkłada wyniki na stos

[10] w innych tłumaczeniach zamiast określenia pieniek można spotkać „łącznik” albo „stopkę klienta”

i wykonuje instrukcję powrotu z procedury (analogicznie jak było to w modelu lokalnego wywołania procedury)
5. pieniek serwera formuje zdjęte ze stosu wyniki do postaci komunikatu odpowiedzi RPC
6. następuje odesłanie wyników do pieńka klienta, który je rozformowuje i odkłada na stos
7. program zdejmuje wyniki działania procedury ze stosu

Widać, iż trzy punkty pierwszego modelu (podanie argumentów, wykonanie funkcji i odebranie wyników) mają swoje odpowiedniki w modelu drugim (odpowiednio punkty 1, 4 i 7) — pozostałe kroki zdalnego wywołania są dla programisty zupełnie niewidoczne co powoduje, że z tego punktu widzenia oba wykonania są faktycznie nierozróżnialne.

1.3.3. Zagadnienia związane z przezroczystością

Aby zaprojektować rzeczywiście przezroczysty mechanizm RPC należy wziąć pod uwagę następujące kwestie:

Przekazywanie argumentów

W przypadku procedur lokalnych argumenty mogą być przekazywane w ogólności na dwa sposoby — przez wartość (ang. *call-by-value*) lub przez odniesienie (ang. *call-by-reference*) czyli w przypadku języka C przez wskaźnik. W przypadku zdalnego wywołania procedur ta druga metoda nie ma oczywiście sensu, ponieważ wskaźnik pokazuje na obiekt znajdujący się w przestrzeni adresowej klienta, do której serwer nie ma dostępu. Typowe protokoły RPC umożliwiają jedynie przekazywanie parametrów przez wartość — przy czym programista musi jawnie określić typ oraz ilość przekazywanych danych (np. nie jest możliwe przesłanie tablicy o nieokreślonej długości).

Wiązanie

Wiązanie (ang. *binding*) — zazwyczaj żąda się od programisty znajomości i jawnego określenia, na którym serwerze mają zostać wykonane procedury — z tym że nie zawsze wymagana jest znajomość adresów sieciowych i portów (dla sieci z rodziny TCP/IP) — czasami jest to jakiś własny sposób adresowania typowy dla danej implementacji RPC (np. w przypadku Sun RPC adres serwera stanowi trójka: IP, numer programu, numer wersji — a numer portu jest znajdowany dynamicznie na podstawie powyższej informacji).

Istnieją także systemy, w których nie trzeba określać miejsca wykonania zdalnej procedury — jest tak w przypadku gdy nie ma znaczenia, na którym z serwerów wykona się procedura natomiast zależy nam np. na równomiernym rozdziale zleceń pomiędzy wszystkie serwery — tzw. balansowanie obciążenie (ang. *load balancing*).

Semantyka RPC w sytuacjach awaryjnych

Wystąpienie błędu w trakcie realizacji zdalnego wywołania procedury wymaga specjalnego obsłużenia. W przypadku lokalnego wywołania procedury

programista nie spodziewa się wystąpienia błędu „serwer niedostępny”, który może wystąpić przy wywołaniu zdalnym. Zatem proste przeniesienie programu działającego lokalnie w środowisko RPC niesie ze sobą ryzyko wystąpienia problemów będących wynikiem braku odpowiedniej obsługi błędów pochodzących z RPC. Konieczne jest stworzenie odpowiednich mechanizmów informowania programisty o sytuacjach wyjątkowych — takich, które wprowadzą jak najmniej różnic między wywołaniem zdalnym a lokalnym.

Dodatkowo należy też zdecydować sie, jaką semantykę wykonania procedury zdalnej chcemy osiągnąć. Wybór zależy od charakterystyki procedur wywoływanych na serwerze a dokładnie od tego czy są on powtarzalne (ang. *idempotent*). Przykładem funkcji powtarzalnej jest obliczenie wartości funkcji sinus lub odczyt aktualnej wartości konta bankowego (możemy takie operacje wykonywać wielokrotnie, to ile razy wykonamy operację nie wpływa na przedmiot operacji). Przykładem funkcji niepowtarzalnej jest dodanie pewnej sumy do stanu konta bankowego — każde jej wykonanie zmienia stan konta i dla jego właściciela nie jest obojętne czy procedura wykona się raz czy dwa razy.

Istnieją trzy różne semantyki zdalnego wywoływania procedur:

dokładnie-raz: Procedura wykonuje się jeden raz, semantyka tak dobra jak niemożliwa do uzyskania w praktyce.

najwyżej-raz: Jeżeli wystąpił błąd to nie wiadomo czy procedura wykonała się czy nie, ale wiadomo, że nie wykonała się dwa lub więcej razy. Typowa sytuacja to taka gdy klient nie otrzymuje odpowiedzi bo serwer przestał działać ale nie wie czy awaria nastąpiła przed czy po wykonaniu zlecenia. Przed ponowieniem żądania klient musi się upewnić co do aktualnej sytuacji.

co-najmniej-raz: Strategia odpowiednia w przypadku procedur powtarzalnych — klient, który nie otrzymuje odpowiedzi ponawia żądanie, co może spowodować powtórne wykonanie zdalnej procedury (gdy odpowiedź nie zaginęła ale była jedynie opóźniona).

Reprezentacja danych

Ze względu na różnorodność zapisu danych w różnych systemach powstaje konieczność następnego ograniczenia swobody programisty w korzystaniu z RPC. Tłumaczenie pomiędzy równoważnymi typami danych nie stanowi problemu — np. zamiana 64-bitowej liczby zmiennoprzecinkowej zapisanej w formacie IEEE na 64-bitową liczbę w formacie IBM[11] i z powrotem, lub zamiana 16-bitowej liczby całkowitej w postaci „młodszy bajt pierwszy” (ang. *little-endian*) nad 16-bitową liczbę w postaci „starszy bajt pierwszy” (ang. *big-endian*). Problem pojawia się w przypadku typów, które nie istnieją w jednym z systemów (np. 128-bitowa liczba całkowita) — standard języka C nie specyfikuje zakresów liczb poszczególnych typów a jedynie relacje między

[11] IBM i IEEE są to dwa formaty zapisu liczb zmiennoprzecinkowych różniące się konstrukcją ale o podobnej funkcjonalności

tymi zakresami, stąd funkcja zwracająca liczbę typu *int* w jednym systemie może zwracać liczbę 64-bitową a w innym 16-bitową.

Większość wymienionych wyżej problemów znalazła swoje mniej lub bardziej udane rozwiązania w istniejących implementacjach RPC, natomiast świadomość ich istnienia pozwala uniknąć niespodzianek przy pracy ze zdalnymi procedurami.

1.3.4. Przykładowe implementacje

Sun RPC

Najpopularniejszą chyba w chwili obecnej implementacją RPC dla platform Unixowych jest opracowany w firmie Sun Microsystems standard Sun RPC. Pakiet oprogramowania składa się logicznie z dwóch części — protokołu (aktualnie oznaczonego jako Wersja 2) oraz standardu zapisu danych XDR[12] (ang. *External Data Representation*).

Popularność systemu Sun RPC związana jest z rozpowszechnieniem zbudowanego na jego bazie zdalnego (sieciowego) systemu plików NFS (ang. *Network File System*). Szerszy opis tego systemu zawarty jest w rozdziale 1.4.2. System NFS a zarazem Sun RPC są dostępne (często domyślnie) w większości systemów Unixowych.

Sun RPC zapewnia semantykę „najwyżej-raz" (w tym także dla bezpołączeniowego protokołu UDP/IP). Dodatkowo umożliwiono także kontrolę tożsamości wywołującego zdalną procedurę — dopuszczalne są trzy sposoby identyfikacji:

- zerowe sprawdzanie tożsamości
- sprawdzanie na zasadach Unixa (prawa użytkowników i grup)
- sprawdzanie na zasadach DES (ang. *Data Encryption Standard* — jest to oparty na kluczach publicznych popularny zestaw metod zapewniających poufność i autentyczność danych)

Xerox Courier

Protokół Xerox Courier powstał w celu zapewnienia komunikacji między Unixem a systemami Xeroxa (podstawowa wersja przeznaczona była dla Unixa 4.3BSD). Implementacja ta oprócz standardowych cech systemu RPC zawiera ciekawe rozwiązanie znacznie poprawiające wydajność wywołań — dodatkowy protokół BDT (ang. *Bulk Data Transfer*) przeznaczony do przesyłania dużych ilości danych (stanowiących argument procedury lub wynik jej działania). Zastosowanie BDT pozwala uniknąć występującego w innych implementacjach sztucznego mnożenia wywołań (tzn. gdy jedno logiczne wywołanie funkcji musi zostać podzielone na kilka odrębnych ze względu na ilość danych, które trzeba wysłać lub odebrać).

[12] dokładna specyfikacja protokołu Sun RPC oraz standardu XDR została opublikowana w postaci dokumentów RFC [6, 7]

Xerox Courier jest zbudowany nad połączeniowym i niezawodnym protokołem SPP będącym odpowiednikiem protokołu TCP zaprojektowanym dla architektury XNS (*Xerox Network System*).

Apollo RPC

Jest to implementacja RPC na potrzeby systemu NCA (ang. *Network Computing Architecture*), która nie wyróżnia się niczym szczególnym. Jedynym wartym zaznaczenia rozwiązaniem jest sposób realizacji semantyki „najwyżej-raz" — w Apollo RPC serwer śledzi kolejne żądania przychodzące od klientów i nie dopuszcza do powtórnego wykonania danego zlecenia.

DCE RPC

Pakiet DCE (ang. *Distributed Computing Environment*) zaprojektowany na zlecenie trzech dużych firm (IBM, DEC i Helwett-Packard) jest nadbudową na istniejące systemy operacyjne (m.in. Unix, VMS) pozwalającą na wygodne i efektywne stworzenie środowiska rozproszonych obliczeń. Na potrzeby tego pakietu powstała specjalna wersja RPC posiadająca kilka ciekawych rozwiązań wyróżniających ją wśród innych implementacji:

- obsługa różnych języków programowania, np. program napisany w języku C może wywołać zdalną procedurę napisaną w Fortranie
- możliwość kontroli tożsamości przy wywołaniu zdalnej procedury (prawa dostępu)
- możliwość zadeklarowania procedury jako powtarzalnej co implikuje zmianę semantyki w odwoływaniu się do tej procedury z „najwyżej-raz" na „co-najmniej-raz"
- rozgłaszanie zdalnego wywołania procedury, tzn. na żądanie klienta zdalna procedura wykonana zostanie na wszystkich dostępnych serwerach (które tę procedurę udostępniają)

Jak widać DCE RPC jest zaawansowaną i rozszerzoną implementacją idei zdalnego wywoływania procedur.

1.4. Przykłady rozproszonych systemów plików

W tym rozdziale zostaną przedstawione główne cechy najważniejszych rozproszonych systemów plików jakie zostały zaprojektowane i wykonane. W opisach główny nacisk położony został na cechy charakterystyczne dla danego systemu oraz ciekawe lub niekonwencjonalne rozwiązania zastosowane przy implementacji.

1.4.1. Unix United

Projekt Unix United jest jedną z najstarszych prób rozszerzenia unixowego systemu plików o funkcjonalność rozproszonego systemu plików. Założeniem projektantów było dodanie odpowiedniej warstwy oprogramowania

do istniejących systemów unixowych (tzw. systemów składowych), tak aby powstały system rozproszony był funkcjonalnie nieodróżnialny od zwykłego.

System jest zbudowany w ten sposób, że całe lokalne drzewo systemu składowego jest widoczne jako podkatalog w drzewie systemu rozproszonego, z tym, że może to być katalog na dowolnym poziomie tego drzewa. Konwencja nazewnicza wygląda tak, że korzeń drzewa lokalnego (a zarazem pewien podkatalog w drzewie systemu rozproszonego) jest dostępny jako tradycyjny znak „/”, natomiast aby dostać się do podkatalogów innych maszyn użytkownik musi jawnie użyć specyfikacji z dwiema kropkami (np. `../maszyna2/usr` oznacza katalog `usr` innego komputera, którego drzewo jest podczepione na tym samym poziomie co nasze drzewo). Zatem użytkownicy są zupełnie świadomi struktury systemu — nie jest zapewniona przezroczystość położenia.

1.4.2. Sun NFS

Sieciowy system plików (ang. *Network File System*) zaprojektowany w firmie Sun umożliwia dostęp do plików zdalnych w dowolnych sieciach (nawet rozległych) — pierwotnie wersja była przeznaczona dla sieci opartych o protokoły UDP/IP i TCP/IP. W systemie NFS zbiór połączonych ze sobą stacji roboczych traktuje się jak zbiór niezależnych maszyn z niezależnymi systemami plików. Celem systemu jest umożliwienie dzielenia zasobów między tymi systemami (dzielenie jest w zasadniczo przezroczyste ale wykonywane na zamówienie). W systemie obowiązują połączenia klient-serwer, jakkolwiek zazwyczaj mamy do czynienia z sytuacją, w której dwie maszyny udostępniają sobie nawzajem pliki — a więc pełnią jednocześnie rolę klientów i serwerów.

Aby dostęp do zdalnego katalogu był przeźroczysty musi zostać wykonana operacja montowania: katalog zdalny montuje się w miejsce katalogu lokalnego systemu plików. Po zakończeniu operacji montowania zamontowany katalog wygląda jak poddrzewo lokalnego systemu plików. Określenie zdalnego katalogu jako argumentu w operacji montowania wykonuje się w sposób nieprzezroczysty — musi być podane jego położenie (tj. nazwę lub adres IP komputera w sieci). Po dokonaniu montażu wszystkie operacje są dokonywane w sposób zupełnie przezroczysty.

Możliwe jest zamontowanie katalogu zdalnego w dowolnym miejscu lokalnego drzewa plików — w szczególności punktem montowania może być korzeń lokalnego systemu plików, co pozwala na korzystanie z kompletnego drzewa systemowego (zdalnego) bezdyskowym stacjom roboczym.

Dopuszcza się montaż kaskadowy. Oznacza to, że serwer może udostępniać innym maszynom katalog lokalny, w którym są zamontowane katalogi zdalne z maszyn trzecich (to że taki sposób dostępu jest możliwy, nie znaczy, że jest on efektywny).

System NFS został zaprojektowany w ten sposób aby umożliwić prostą implementację w dowolnym środowisku i współpracę heterogenicznych systemów. Cecha ta została uzyskana dzięki zastosowaniu niezależnych od

systemów operacyjnych standardów zdalnego wywoływania procedur oraz zewnętrznej reprezentacji danych XDR[13].

Protokół NFS

Protokół NFS dostarcza zestawu wywołań procedur zdalnych — operacji plikowych. Umożliwiają one następujące działania:

- szukanie pliku w obrębie katalogu
- czytanie zbioru wpisów katalogowych
- manipulowanie katalogami
- manipulowanie plikami (dostęp do niektórych atrybutów)
- czytanie i modyfikacja plików

Ominięcie operacji otwarcia i zamknięcia pliku jest zamierzone — serwery systemu NFS są bezstanowe. Ponieważ NFS nie dostarcza mechanizmów sterowania współbieżnością (zajmowania plików itp.) a wielkość jednorazowo przesyłanych danych jest ograniczona (zależnie od implementacji i zastosowanych protokołów ograniczenie waha się od 1KB do 8KB) to dane zapisywane przez dwóch klientów mogą zostać przemieszane. Architekci systemu NFS zalecają użytkownikom chcącym korzystać ze współdzielenia plików stosowanie zewnętrznych (nie wbudowanych w NFS) mechanizmów koordynacji.

O dość swobodnym podejściu do semantyki Unixowej w systemie NFS świadczy to, że stosuje się w nim czytanie z wyprzedzeniem i zapis z opóźnieniem jednocześnie, nawet gdy plik jest otwarty współbieżnie w dwóch kolidujących trybach (odczyt i zapis). W celu zredukowania ryzyka wprowadzono kontrolę czasu i tak odpowiednio bloki danych są uaktualniane w ciągu co najwyżej 3 sekund natomiast katalogi w ciągu co najwyżej 30 sekund — tak więc nowo utworzony w katalogu plik może być widoczny dla innych systemów dopiero po trzydziestu sekundach. Głównym założeniem projektantów było uzyskanie dużej funkcjonalności oraz maksymalizacja wydajności. Testy systemu NFS z modułami serwera i klienta umieszczonymi w jądrze systemu Unix[14] wykazały różnice wielkości 20% w porównaniu do lokalnego systemu plików (testy były wykonywane na mało obciążonych serwerach, doświadczenie pokazuje niską skalowalność systemu — opóźnienia rosną bardzo szybko wraz ze wzrostem obciążenia serwerów). Funkcjonalność i wydajność były tymi cechami, które przyczyniły się do rozpowszechnienia systemu NFS. Jest to chyba najbardziej popularny system rozproszonego dostępu do zasobów (dostarczany z większością dzisiejszych unixowych systemów operacyjnych).

1.4.3. Andrew

System AFS (ang. *Andrew File System*) powstał w Carnegie-Mellon University i jest jednym z najciekawszych rozproszonych systemów plików. Jedną z najbardziej znamiennych jego cech jest skalowalność (potrafi standardowo obsłużyć kilka tysięcy stacji roboczych – w systemie rozdziela się pojęcie

[13] Sun RPC i XDR zostały omówione w rozdziale 1.3.4

[14] umieszczenie modułów w jądrze znacznie przyspiesza pewne operacje

stacji roboczej czyli klienta i serwera). Bazą dla systemu jest Unix w wersji 4.2BSD.

Przestrzenie nazw plików klientów obejmują dwa obszary — lokalną przestrzeń plików i wspólną (dzieloną) przestrzeń plików. Dzielona przestrzeń adresowana jest przez jednolitą i przezroczystą względem położenia strukturę nazewniczą.

Wszystkie stacje robocze i serwery są pogrupowane w logiczne grupy (grona) odpowiadające zazwyczaj pewnym segmentom sieci lokalnych. Grona te są połączone siecią rozległą. Dąży się do tego aby klienci w miarę możliwości komunikowali się tylko z lokalnymi (względem grona) serwerami, natomiast między gronami łączność zachowują serwery.

Za kluczowy mechanizm działań na plikach zdalnych przyjęto duże pamięci podręczne klientów o dużej (co najmniej 64KB) ziarnistości. Założeniem było też wyposażenie stacji roboczych w dyski — co umożliwia stałość pamięci podręcznych (jest to zgodne z obserwacją, że pliki są rzadko modyfikowane — czyli pamięć podręczna nie musi być często aktualizowana). Pliki przesyłane są pomiędzy serwerem i klientem zawsze w całości, uaktualnianie następuje przy zamykaniu pliku.

Całość opiera się na obserwacji, że dany użytkownik (przypisany do stacji roboczej) najczęściej używa małej części z dostępnych programów, lecz zdarza mu się także sięgnąć do programów zwykle nie używanych. Po pewnym czasie pracy z systemem AFS w pamięci podręcznej stacji (na jej lokalnym dysku) znajdą się właśnie te najczęściej używane programy i korzystanie z nich nie będzie generowało ruchu w sieci (także dlatego, że aktualizacja pamięci podręcznej zbudowana jest w oparci o przywołania (ang. *callback*) — gdzie stroną aktywną jest serwer). Kontakt z serwerem będzie konieczny jedynie w momencie konieczności aktualizacji oprogramowania w pamięci podręcznej lub skorzystania z nieużywanego dotychczas programu. Warto zauważyć również, że mimo iż powyższe rozumowanie jest prawdziwe dla większości klientów, to pamięci różnych użytkowników będą różne — ich zawartość będzie zależała od profilu działalności danego klienta. Zachodzi swoista automatyczna optymalizacja według indywidualnych upodobań.

Dzielona przestrzeń nazw systemu AFS składa się z jednostek zwanych tomami (inaczej woluminami). Każdy serwer posiada informacje co do położenia tomów i ich replik. Zwielokrotnianie odbywa się na poziomie tomów i jest automatyczne — takie podejście pozwala na optymalizację odwołań do plików, gdyż adresowanie wewnątrz tomów pozostaje stałe, niezależnie od położenia danego tomu, które może się zmieniać w wyniku działania replikacji.

System AFS jest jednym z najbardziej zaawansowanych i zarazem najwydajniejszych rozproszonych systemów plików. Część jego rozwiązań została zaadoptowana w projekcie systemu DGFS (opisanego w drugim rozdziale ninejszej pracy).

1.4.4. DCE DFS

Rozproszony system operacyjny DCE (projekt powstały w *Open Systems Foundation*) implementuje rozdzielone logicznie usługi katalogowe i usługi plikowe. Projekt systemu powstał jako komercyjna implementacja systemu AFS (opisanego wyżej) a zarazem jako jego logiczne rozwinięcie.

Usługi katalogowe umożliwiają dostęp przez nazwę do wszystkich zasobów występujących w ramach systemu. W tym przypadku oprócz plików oznacza to m.in. użytkowników, serwery, dane dotyczące bezpieczeństwa i usługi[15]. W systemie wszystkie zasoby grupowane są w logiczne komórki (ang. *cells*) — komórki zawierają własne lokalne serwisy nazewnicze (obejmujące tylko przestrzeń danej komórki). Oprócz nich istnieją mechanizmy całościowe — tzw. globalne usługi katalogowe. Sposób działania i składnia nazw jest nieco zbliżona do systemu NDS (*Novell Directory Services*), komórki są umieszczone w drzewie, którego poziomy odzwierciedlają rzeczywistą strukturę logiczną (kraj, położenie, organizacja, oddział organizacji (ang. *OU, organization unit*), grupa, użytkownik itd.). Klienci posługują się nazwami według logicznej struktury powiązań a nie według struktury sieci, która często ma się nijak do rzeczywistości. Na przykład ścieżka `/C=US/O=UCLA/OU=CS/user/ann/x` oznacza odwołanie do pliku `x` będącego w posiadaniu użytkownika `ann`, który pracuje na wydziale informatyki (`OU=CS` oznacza *Organization Unit = Computer Science*) uniwersytetu w Los Angeles (`O=UCLA` oznacza *Organization = University of California, Los Angeles*) w Stanach Zjednoczonych (`C=US` oznacza *Country = United States*).

Usługi plikowe w systemie DCE, określane po prostu jako DFS (ang. *Distributed File System*), są zaprojektowane jako rozproszony system plików rozpięty nad sieciami o zasięgu globalnym. Zbudowany jest z dwóch części — lokalnej (zwanej Epizodem) i globalnej. W DFS każda maszyna jest tak serwerem swoich plików jak i klientem wobec dowolnej innej maszyny systemu. Stosuje się intensywne zwielokrotnianie plików (m.in. na zwielokrotnianiu opiera się dystrybucja nowych wersji programów przeznaczonych do wykonywania w systemie). Lokalna część systemu jest zgodna ze standardowym drzewem Unixa i może je zastępować. Implementowana jest semantyka zgodna z Unixową.

1.4.5. Amoeba

System rozproszony Amoeba został zaprojektowany przez zespół pracujący pod kierownictwem jednego z największych specjalistów w dziedzinie przetwarzania rozproszonego i równoległego — A. S. Tanenbauma (autora m.in. [1]). Dla przeciętnego użytkownika praca w systemie Amoeba nie różni się od pracy w zwykłym Unixie — jest to pełna emulacja systemu z podziałem czasu. Projekt nie jest ograniczony do obsługi plików lecz realizuje

[15] niektóre usługi też mogą być potraktowane jak pliki, typowym przykładem jest tu unixowy katalog `/proc` — pliki w tym katalogu odzwierciedlają procesy w systemie i operacje na tych plikach przenoszą się na operacje na procesach

pełne rozproszenie procesów w systemie, współdzielenie pamięci operacyjnej, specjalizowane protokoły sieciowe i inne elementy składające się na rozproszony system operacyjny, tu jednak przedstawiony zostanie jego fragment zajmujący się obsługą zasobów. Opis będzie dotyczył wersji trzeciej systemu.

Standardowy system plików systemu Amoeba składa się z trzech serwerów: serwera pocisków (oryginalnie *bullet server*, w literaturze jest to tłumaczone też jako „serwer-kulomiot") obejmującego magazyn plików, serwera katalogów i serwera zwielokrotnień. Konstrukcja serwera pocisków (plików) jest znacząco różna od spotykanych w innych systemach. Podstawową jego cechą jest to że pliki są stałe — to znaczy nie możliwa jest modyfikacja pliku a jedynie jego usunięcie i utworzenie na jego miejscu nowego o tej samej nazwie. Upraszcza to znacząco zwielokrotnianie i konstrukcję pamięci podręcznych. Pomysł takiego serwera plików opiera się na założeniu, że klient tworzy plik we własnej pamięci, a następnie przesyła go za pomocą jednego wywołania procedury zdalnej do serwera. Aby pozwolić klientom o małej pamięci modyfikację dużych plików dodano możliwość przesyłania danych we fragmentach, które jednak czekają na serwerze i dopiero po skompletowaniu wszystkich kawałków plik oryginalny zostaje zastąpiony przez nowy złożony z tychże fragmentów.

Do obsługi struktury nazewniczej jest wydzielony osobny serwer katalogów. Odwzorowuje on nazwy czytelne dla użytkownika na odpowiednie informacje pozwalające odnaleźć dany plik na serwerze plików. Katalogi (w odróżnieniu od plików) nie są stałe. Można dodawać i usuwać z nich wpisy o plikach. Serwer katalogów jest konstrukcją na tyle ogólną, że może oprócz adresowania plików służyć do adresowania dowolnych innych obiektów istniejących w systemie — tak fizycznych (urządzenia, procesory) jak i logicznych (dostęp do dzielonej pamięci).

Serwery zwielokrotnień w systemie Amoeba działają na zasadzie leniwego zwielokrotniania i tworzą dobre uzupełnienie całego systemu. Opisana wyżej semantyka plików stałych została wybrana między innymi ze względu na możliwość efektywnej implementacji zwielokrotniania.

1.4.6. Sprite

System Sprite jest eksperymentalnym systemem rozproszonym opracowanym na uniwersytecie stanu Kalifornia w Berkeley[16]. Główny nacisk położono w nim na badania zagadnień sieciowych systemów plików, wędrówki procesów oraz wysokowydajnych systemów plików. Został zrealizowany, wdrożony i z powodzeniem używany.

Konstrukcja systemu wygląda następująco: istnieje kilka dużych i silnych serwerów wyposażonych w odpowiednio wielkie dyski oraz wiele bezdyskowych stacji roboczych (wykorzystujących własną pamięć do przechowywania podręcznego). System plików jest podobny do tradycyjnego systemu w Uni-

[16] z tego samego uniwersytetu pochodzi jedna z dwóch głównych linii rozwojowych systemu Unix, tzw. BSD Unix (ang. *Berkeley Software Distribution*), od której wywodzą się m.in. popularne dziś systemy FreeBSD, OpenBSD i NetBSD

xie — cały system to jedno drzewo unixowe, obejmujące wszystkie zasoby (pliki i urządzenia sieci), czyniąc je jednakowo przezroczyście dostępnymi na każdej stacji roboczej. Przezroczystość w systemie Sprite jest pełna — nie ma możliwości określenia położenia pliku w sieci na podstawie jego nazwy.

System Sprite wymusza spójność plików dzielonych. Każdemu odwołaniu do systemu za pomocą funkcji *czytaj* zapewnia się otrzymanie w pełni aktualnych danych. Emulowany jest system Unix z podziałem czasu.

Istotnym aspektem systemu Sprite jest poszerzone zastosowanie technik pamięci podręcznej. Ma ona następujące cechy:

- jako pamięć podręczna wykorzystywane jest tylko pamięć operacyjna (stacje są głównie bezdyskowe)
- jednostką organizacji pamięci podręcznej jest blok (a nie plik), aktualna wielkość bloku to 4KB
- pamięć jest stosowana jednocześnie do przechowywania czytanych bloków, jak i do przechowywania bloków zapisywanych z opóźnieniem
- wielkość pamięci podręcznej zmienia się dynamicznie w zależności od zapotrzebowania
- spójność pamięci podręcznej jest nadzorowana przez serwer (który odnotowuje otwarcia plików i próby zapisu, informuje o dezaktualizacji pamięci stacji roboczej itd.)

1.4.7. Locus

System Locus został zbudowany z założeniem stworzenia kompletnego rozproszonego systemu operacyjnego (a nie jedynie systemu rozproszonego udostępniania zasobów). Nie jest on dodatkiem lub rozszerzeniem istniejącego systemu lecz całkowicie nowym rozwiązaniem (oznacza to m.in. nowe, zaprojektowane od podstaw jądro systemu).

Podobnie jak w systemie Sprite mamy tu do czynienia z jedną kompletną drzewiastą strukturą nazewniczą obejmującą wszystkie zasoby (pliki, katalogi i urządzenia) dostępne na wszystkich maszynach w systemie. Jest to system w pełni przezroczysty — na pierwszy rzut oka nie można odróżnić struktury Locusa od standardowego drzewa Unixa.

W systemie Locus jednemu plikowi może odpowiadać zbiór kopii rozproszonych na różnych stanowiskach. Wewnętrzne mechanizmy systemu są odpowiedzialne za aktualność wszystkich kopii i dostęp do najnowszych wersji. Modyfikacje plików odbywają się na zasadzie uaktualniania kopii podstawowej.

Semantyka w systemie naśladuje standardową semantykę Unixową ale istnieje również alternatywny mechanizm blokowania plików (i ich części) realizujący semantykę zbliżoną do semantyki transakcji.

Duży nacisk położono na działanie w warunkach odłączenia i awarii. Dopóki dostępne są kopie pliku, dopóty zapewnione jest czytanie z najnowszej z nich. Istnieją też wyrafinowane mechanizmy aktualizacji danych po powrocie z odłączenia.

Struktura nazewnicza opiera się na grupach plików (zwanych kontenerami). Każde stanowisko ma pełny obraz logicznej struktury nazw i wypełniających ją kontenerów). Adresowanie (a co za tym idzie pamięci podręczne i replikacja) obejmuje właśnie kontenery. Dodatkowo istnieją mechanizmy tłumaczące odpowiednio odwołania do plików specjalnych (np. urządzeń w katalogu `/dev`) i plików wykonywalnych (system Locus może obejmować komputery heterogeniczne pod względem architektury więc musi istnieć odpowiednia liczba wersji plików wykonywalnych aby maszyny uruchamiały programy odpowiednie dla swojej architektury).

Rozdział 2

DGFS jako przykład w pełni rozproszonego systemu plików

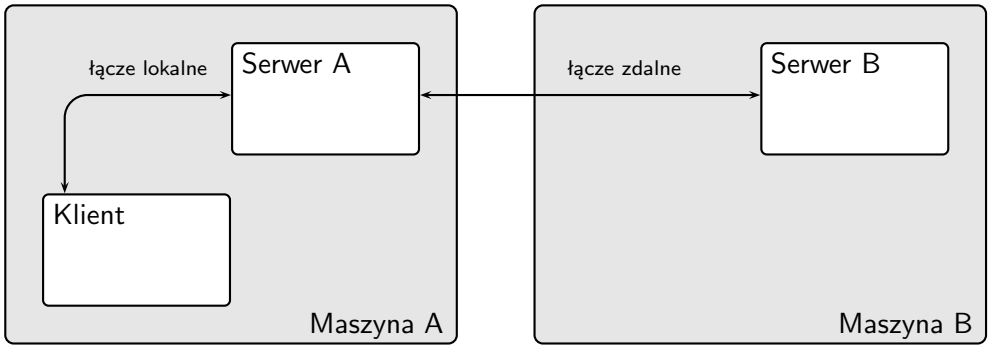
2.1. Projekt

Wiadomości przedstawione w poprzednim rozdziale były inspiracją i punktem wyjścia do opracowania projektu rozproszonego systemu udostępniania zasobów, który łączyłby najciekawsze cechy istniejących systemów, biorąc zarazem pod uwagę rozważania teoretyczne i wnioski płynące z badań prowadzonych nad lokalnymi systemami plików. Nacisk przy tworzeniu projektu został położony bardziej na stronę badawczą niż praktyczną — przy wyborze konkretnych rozwiązań faworyzowane były warianty nowatorskie, ciekawe, o wysokiej funkcjonalności. Rozwiązania sprawdzone i dobrze poznane były wybierane wtedy, gdy miały one znaczącą przewagę jeżeli chodzi o wydajność lub funkcjonalność.

2.1.1. Podstawy projektu

System DGFS jest to sieć serwerów uruchomionych na maszynach połączonych siecią umożliwiającą adresowanie przy pomocy protokołu IP. Każda maszyna posiada jeden serwer, który zajmuje się między innymi udostępnianiem plików ulokowanych na tej maszynie oraz udostępnianiem zasobów całego systemu klientom lokalnym — z założenia każda maszyna jest to komputer działający pod kontrolą wieloużytkownikowego systemu unixowego. Taka postać systemu oznacza, że aby klient mógł korzystać z zasobów systemu, to na tej samej maszynie musi być uruchomiony serwer DGFS — nie ma możliwości zdalnego korzystania z serwerów. Schemat połączeń został przedstawiony na rys. 2.1.

Dostęp do systemu odbywa się poprzez bibliotekę funkcji języka C, za pomocą której można korzystać z usług systemu. Dopiero oprogramowanie napisane przy użyciu tej biblioteki będzie przeznaczone dla zwykłego użytkownika. Opisywane poniżej cechy dotyczą dostępu poprzez tą bibliotekę (czyli określenie „klient” oznacza w tym przypadku programistę). Oznacza to, że nawet jeżeli któraś z cech nie jest zapewniona na poziomie biblioteki, to może ona zostać dodana przez programistę (na przykład obsługa błędów przekroczenia czasu czekania (ang. *timeout*) — program używający biblioteki może



Rysunek 2.1: Schemat połączeń w systemie DGFS

automatycznie ponawiać żądania, co nie będzie widoczne dla znajdującego się poziom wyżej użytkownika).

2.1.2. Nazewnictwo

System w pełni realizuje przezroczystość i niezależność położenia. Pliki umieszczone w systemie są dostępne poprzez jednolite dla wszystkich klientów drzewo katalogów. Każdy klient niezależnie od tego, do którego serwera jest podłączony zawsze używa takich samych nazw plików. Nie jest także świadomy tego czy wykorzystany został mechanizm pamięci podręcznej lub replikacji, ani tego gdzie w sieci znajdował się w momencie wykonania żądania dany plik.

Przezroczystość jest uzyskana dzięki wielowarstwowym wywołaniom — każde żądanie klienta przechodzi przez kilka poziomów, klient kontaktuje się jedynie z własnym serwerem i nie wie jakie działania są przez serwer podejmowane w celu realizacji żądania. Nie wie zatem czy serwer kontaktuje się z innymi serwerami (nie mówiąc o tym, że nie wie z którymi). W odpowiedzi na swoje żądanie klient po prostu otrzymuje dostęp do pliku, którego nazwę wyspecyfikował.

2.1.3. Katalogi

Struktura nazewnicza jest w postaci tradycyjnego drzewa katalogów. Każdy katalog jest przypisany na stałe do jednego z serwerów. Serwer obsługujący dany katalog (inaczej: „właściciel danego katalogu") jest odpowiedzialny za realizowanie wszystkich żądań dotyczących tego katalogu jak i jego zawartości. Między innymi zna stan i położenie (w sieci) plików w danym katalogu.

Każdy serwer może obsługiwać dowolną liczbę katalogów, natomiast dany katalog jest przypisany do jednego serwera. Katalogi tworzone w czasie pracy przez klientów są automatycznie przypisywane do tych serwerów, do których są oni podłączeni.

Serwer nie musi znać właścicieli wszystkich katalogów w drzewie (nawet nie musi znać całej struktury drzewa). Jedyne informacje jakie są mu potrzeb-

ne do pracy to adresy właścicieli katalogów nadrzędnych względem własnych katalogów, oraz adresy właścicieli podkatalogów położonych wewnątrz własnych katalogów. Informacje o innych fragmentach drzewa znajdywane są dynamicznie (poprzez wymianę danych z innymi serwerami) wtedy gdy jest taka potrzeba — klient żąda dostępu do pliku położonego w katalogu, którego właściciela serwer nie zna.

2.1.4. Pliki; Pamięć podręczna i zwielokrotnianie

Pliki w przeciwieństwie do katalogów nie mają wyznaczonych właścicieli i mogą występować w systemie w wielu równoważnych kopiach. Zastosowano mechanizm łączący cechy pamięci podręcznej i zwielokrotniania bez kopii głównej.

Właściciel katalogu, w którym znajduje się plik posiada listę serwerów, które posiadają kopię danego pliku. Jeżeli serwer, który takiej kopii nie posiada, zgłosi żądanie pobrania tego pliku, to właściciel katalogu udostępnia mu listę posiadaczy kopii. Wtedy serwer może pobrać ten plik od dowolnego serwera z listy (plik zostaje zapisany w lokalnym systemie plików). On sam zostaje na tę listę dopisany, ponieważ posiada kopię pliku — odpowiada to mechanizmowi automatycznego zwielokrotniania pliku. Klienci serwerów, które posiadają kopie jakiegoś pliku, otrzymują do niej dostęp natychmiast — trudno w tym momencie określić czy mamy do czynienia z pamięcią podręczną czy ze zwielokrotnieniem.

2.1.5. Semantyka współdzielenia plików

Zgodnie z założeniem, że pliki są rzadko współdzielone przy zapisie, zastosowano dość drastyczną semantykę. Jest to połączenie semantyki sesji z semantyką transakcji. Pliki otwierane do modyfikacji są blokowane (nikt inny nie może wtedy otworzyć takiego pliku do modyfikacji). Zmiany są widoczne dopiero po zamknięciu pliku — nie ma tu jednak typowej dla transakcji operacji *cofnij* (ang. *rollback*), która odwoływałaby zmiany przeprowadzone na pliku. Nie można też jedną transakcją objąć zmian w kilku plikach (ale oczywiście można otworzyć do zapisu kilka plików jednocześnie). Oczywiście w trakcie modyfikowania pliku przez jednego klienta inni nie mogą tego uczynić, ale mogą czytać dany plik — dostają wtedy postać pliku sprzed rozpoczęcia modyfikacji. Po zakończeniu modyfikacji klienci, którzy czytają ze starych kopii, nadal mają do nich dostęp — nową zawartość zmodyfikowanych plików odczytają dopiero przy następnym otwarciu pliku. Nie jest to więc klasyczna semantyka sesji, w której zmiany w plikach są widoczne dla klientów czytających natychmiast po zakończniu modyfikacji.

Blokowaniem i aktualizacją zajmuje się właściciel katalogu, w którym znajduje się dany plik. Oznacza to konieczność kontaktu z właścicielem katalogu przy otwieraniu pliku do modyfikacji (nawet w przypadku posiadania lokalnej kopii pliku). Ponieważ, jak opisano wyżej, katalog ma jednego właściciela, nie ma problemu z koordynacją blokowania — odbywa się ono

w jednym miejscu. Aktualizacja odbywa się z inicjatywy serwera, właściciela katalogu. Ponieważ nie ma wydzielonej kopii głównej zmieniony plik pozostaje na serwerze, na którym był modyfikowany, natomiast właściciele pozostałych kopii (w tym momencie nieaktualnych) otrzymują informacje o dezaktualizacji kopii i usuwają je. Później jeżeli ich klienci będą potrzebować dostępu do pliku (do czytania lub modyfikacji), będą musieli go ponownie pobrać — nastąpi ponowne rozprzestrzenienie się kopii w systemie.

2.1.6. Awarie; Działanie w odłączeniu

Zastosowany mechanizm zwielokrotniania powoduje, że nawet jeżeli serwer — właściciel pewnego katalogu ulegnie awarii, to klienci innych serwerów mogą mieć dostęp do plików umieszczonych w tym katalogu o ile serwery posiadające ich kopie są dostępne (oczywiście możliwy jest jedynie odczyt, gdyż brak właściciela katalogu uniemożliwa przeprowadzenie procedury otwarcia pliku do modyfikacji). Co więcej, ze względu na sposób działania systemu (podobny do działania systemu Andrew) — jeżeli serwer działał wystarczająco długo, to kopie plików najczęściej używanych przez jego klientów znalazły się na tym serwerze i klienci mogą pracować nawet w sytuacji całkowitego odcięcia serwera od reszty systemu.

2.1.7. Skalowalność i wydajność

Wydaje się, że wybór opisanej wyżej semantyki i sposobu replikacji plików jest optymalny jeżeli chodzi o ograniczanie obciążenia sieci. Widać to, jeżeli weźmie się pod uwagę z jednej strony to, że większość plików modyfikowana jest sporadycznie, a z drugiej, że jedyny ruch w sieci powstaje w momencie aktualizacji zmienionego pliku (która zresztą nie jest przeprowadzana przez wszystkie serwery jednocześnie, ale w miarę powstawania zapotrzebowania).

Dodatkowo stworzone zostały oddzielne rozwiązania dla transferu plików (opisane w rozdziale 2.2.8) oraz wyposażono serwery w odpowiednią (modyfikowalną) ilość niezależnych „wejść” i „wyjść” służących do kontaktu z klientami i innymi serwerami — tak aby żądania wydane przez jednych nie wymuszały czekania na innych. Wydaje się, że właśnie obciążenie sieci oraz wzajemne blokowanie klientów jest tym co ogranicza najbardziej skalowalność w istniejących rozproszonych systemach plików. W systemie DGFS oba te powody zostały możliwie zredukowane. Także doświadczenia systemu Andrew pokazują, że podobne rozwiązania wykazywały się dużą skalowalnością.

2.2. Implementacja

Podstawowym założeniem fazy implementacji było oparcie tworzonego systemu o istniejące systemu unixowe. Serwer jest programem działającym w trybie użytkownika (a nie jako fragment jądra systemu operacyjnego). Jedną z ważniejszych cech tworzonego kodu jest jego przenośność w obrębie różnych wersji unixa.

2.2.1. Struktura systemu

Serwer systemu DGFS jest zbiorem oddzielnych programów, nazywanych dalej modułami. Każdy z nich pełni oddzielną funkcję (zakres obowiązków jest ściśle rozdzielony) lecz współpracuje z pozostałymi modułami. Dokładna struktura systemu jest przedstawiona na rysunku 2.2. System uruchamiany jest poprzez wystartowanie modułu nadrzędnego (wszystkie moduły zostaną szczegółowo opisane w dalszej części). Parametry konfiguracyjne określające warunki i sposób pracy systemu są zapisane w odpowiednim pliku lokalnego systemu plików danej maszyny[1].

Ważnym pojęciem w systemie DGFS jest *magazyn plików*. W sensie logicznym jest to miejsce gdzie są trzymane kopie plików, a także miejsce gdzie tworzone są pliki tymczasowe i inne. Każdy plik w magazynie jest identyfikowany poprzez unikalny identyfikator (ang. *UFID, Unique File IDentifier*). UFID jest wykorzystywany przy identyfikacji plików w komunikacji między modułami co jest konieczne w sytuacji gdy użycie nazwy (systemu DGFS) nie jest jednoznaczne — sytuacja taka ma miejsce w czasie modyfikacji pliku: w momencie gdy jeden z klientów modyfikuje plik, inni muszą mieć dostęp do starej wersji, czyli jeden logiczny plik istnieje w dwóch wersjach, starej i nowej, aż do czasu zakończenia modyfikacji. Identyfikatory są generowane przez opisany niżej moduł obsługi plików.

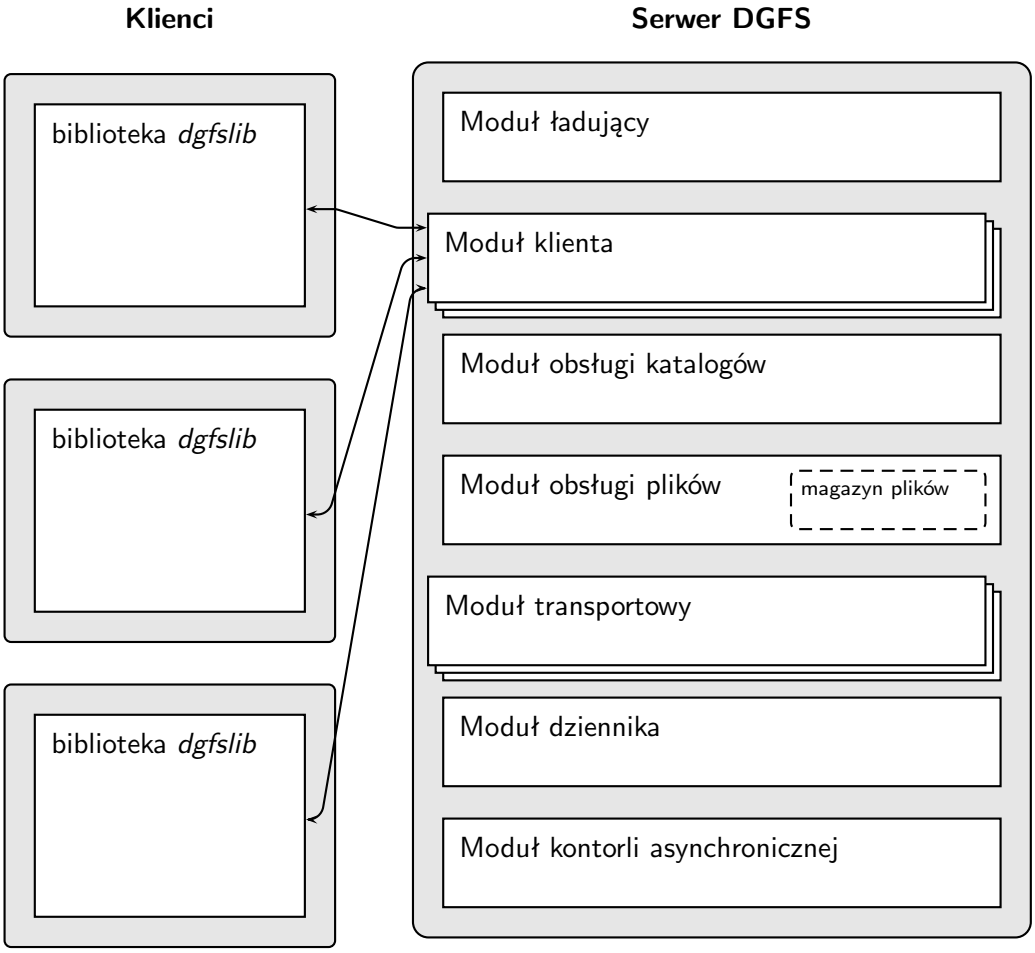
W sensie praktycznym magazyn plików jest pewnym katalogiem lokalnego systemu plików. Nazwy plików w tym katalogu są tworzone na podstawie unikalnych identyfikatorów (np. plik o nazwie `dgfs1234.dat` jest plikiem o identyfikatorze 1234, jego nazwa w systemie DGFS jest znana modułowi obsługi plików). Takie rozwiązanie pozwala na dostęp do plików systemu DGFS za pomocą standardowych unixowych funkcji wejścia-wyjścia. Ważne jest, że dostęp taki (czyli np. otwarcie pliku) realizuje biblioteka programistyczna (także opisana poniżej) a nie sam program klienta. Klient nie zna wartości identyfikatorów i posługuje się nazwami systemu DGFS.

2.2.2. Biblioteka programisty

Dostęp do wszystkich funkcji systemu DGFS odbywa się poprzez interfejs biblioteki języka C. Głównym zadaniem biblioteki jest przetłumaczenie żądań programisty na zdalne wywołania procedur w systemie DGFS, a dokładnie w jego module klienta (opisanym niżej w rozdziale 2.2.4). Biblioteka daje możliwość napisania oprogramowania wykorzystującego możliwości systemu bez zagłębiania się w szczegóły techniczne dotyczące jego funkcjonowania.

Oczywiście programista powinien znać pewne zasady rządzące systemem i filozofię jego działania. Na przykład różnicą, o której trzeba pamiętać, jest semantyka współdzielenia plików — korzystając z plików poprzez bibliotekę DGFS należy otwierać je do zapisu na jak najkrótszy okres czasu — gdyż

[1] zgodnie z konwencją unixową domyślnie jest to plik `/etc/dgfs`



Rysunek 2.2: Schemat systemu DGFS w obrębie jednej maszyny

w tym czasie są one zablokowane a zmiany wprowadzane do nich nie są widoczne dla innych do czasu zamknięcia sesji z plikiem[2].

Interfejs

```
typedef int DGFS_DIR;
```

Większość funkcji bibliotecznych wymaga użycia deskryptorów plików i katalogów. Deskryptor katalogu jest zdefiniowany jak powyżej natomiast w przypadku plików używany jest zwykły unixowy deskryptor pliku (`FILE`). Jest to spowodowane wykorzystaniem zwykłych funkcji standardowego interfejsu wejścia-wyjścia (`stdio`).

```
int dgfs_connect(void);
```

Funkcja inicjująca pracę biblioteki. Zajmuje się alokacją pamięci, przygotowuje komunikację z systemem, ustawia odpowiednie parametry itd. Po jej udanym zakończeniu można rozpocząć korzystanie z usług systemu DGFS.

[2] różne warianty współdzielenia plików są opisane w rozdziale 1.1.4

`int dgfs_disconnect(void);`
Wywołanie tej funkcji powoduje odłączenie od systemu DGFS. Pierwszym zadaniem funkcji jest „kulturalne” zakończenie współpracy z systemem — zamknięcie otwartych plików (i katalogów), jeżeli takowe są, oraz odłączenie kanałów komunikacyjnych. Drugie to zniszczenie stworzonych w czasie pracy struktur danych i zwolnienie zajmowanej pamięci.

`DGFS_DIR dgfs_opendir(char *dirname);`
Funkcja otwierająca do przeglądania katalog systemu. Jako argument przyjmuje pełną ścieżkę katalogową. Ostatni element może zawierać symbole wieloznaczne (obsługiwane są wyrażenia regularne). Konieczność podawania pełnej (bezwzględnej) ścieżki wynika z braku „katalogu bieżącego”, który mógłby być punktem dla nazw względnych. Funkcja zwraca deskryptor katalogu, który używany jest przy wywoływaniu dwóch następnych funkcji.

`int dgfs_readdir(DGFS_DIR dir, dirent *buffer);`
Funkcja odczytuje jeden wpis katalogowy z katalogu określonego deskryptorem `dir`. Struktura `dirent` jest standardową unixową strukturą zawierającą oprócz nazwy podstawowe atrybuty pliku takie jak typ i długość. Wartość zwracana informuje czy był to ostatni wpis z danego katalogu.

`int dgfs_closedir(DGFS_DIR dir);`
Funkcja kończąca odczyt katalogu — informuje system DGFS o zakończeniu tej operacji dla danego katalogu.

`int dgfs_erase(char *filename);`
Funkcja kasująca plik (określony przez nazwę poprzedzoną pełną ścieżką). Zwraca informację czy kasowanie się powiodło.

`int dgfs_rename(char *oldname, char *newname);`
Funkcja zmieniająca nazwę pliku, stara nazwa (`oldname`) zostaje zastąpiona nową (`newname`) o ile jest to możliwe. Podobnie jak w przypadku poprzednich funkcji nazwy muszą zawierać pełne ścieżki.

`FILE *dgfs_open(char *filename, char *mode);`
Funkcja służy do otwarcia pliku o podanej nazwie — jest to odpowiednik funkcji `fopen` z interfejsu `stdio`. O trybie otwarcia decyduje parametr `mode`, którego znaczenie jest podobne jak dla funkcji `fopen`: „`r`” oznacza otwarcie pliku do odczytu, natomiast „`w`” otwarcie do zapisu. Podane tryby są informacją dla biblioteki, gdyż faktycznie plik jest udostępniany jako otwarty w trybie „`r`” (odczyt) i „`r+`” (modyfikacja). Dodatkowo mogą być użyte modyfikatory „`t`” (tryb tekstowy) i „`b`” (tryb binarny). Funkcja zwraca unixowy deskryptor pliku, który następnie może być użyty ze wszystkimi funkcjami interfejsu `stdio` (`fread`, `fgets`, `fprintf` itd.) — jedynie zamknięcie pliku powinno być wykonane poprzez niżej opisaną funkcję `dgfs_close`.

`int dgfs_close(FILE *f);`
Funkcja zamyka plik według przekazanego deskryptora oraz informuje o tym system, który (jeżeli plik był otwarty do zapisu) musi wykonać szereg czynności mających na celu aktualizację informacji w całym systemie.

2.2.3. Nadrzędny moduł uruchomieniowy

System DGFS został pomyślany jako zespół oddzielnych lecz współpracujących modułów — każdy z nich jest oddzielnym programem i działa jako osobny proces (w sensie unixowym). Istnieje też program główny, którego zadaniem jest uruchomić poszczególne moduły i przekazać im odpowiednie parametry konfiguracyjne. W dalszej części będzie on określany mianem nadrzędnego demona[3] DGFS, w skrócie *dgfsd*.

W trakcie działania systemu moduł ten nie pełni żadnej roli, jest jednak rodzicem grupy procesów składającej się z pozostałych modułów (unixowe polecenie *pstree* pozwala obejrzeć strukturę procesów, na rys. 2.3 przedstawione jest drzewo procesów systemu DGFS uruchomionego w systemie FreeBSD Unix). W związku z tym zostały mu powierzone dodatkowe zadania — zatrzymanie i restart systemu na życzenie administratora. W tym przypadku komunikacja odbywa się przy użyciu unixowych sygnałów.

2.2.4. Moduł klienta

Moduł klienta pełni funkcję pośrednika pomiędzy biblioteką w programie klienta a systemem DGFS (jego głównymi modułami). Moduł ten nie przechowuje żadnych informacji o klientach (informacje o otwartych plikach są przechowywane przez moduł obsługi plików).

Szczególną cechą modułu klienta jest to, że może zostać uruchomiony wielokrotnie — ilość uruchomionych modułów klienta jest zależna od ilości klientów w systemie (z tym, że nie oznacza to równości; bez przeprowadzania testów można stwierdzić, że rozsądną ilością jest jeden moduł klienta na trzech do dziesięciu potencjalnych klientów). Nie stosuje się wyszukiwania wolnych modułów klienta lecz algorytm *round-robin* — klient pierwsze żądanie wysyła do pierwszego modułu, drugie do drugiego i tak do ostatniego a kolejne otrzymuje znów pierwszy. Wydaje się, że komplikacja wynikła z wyszukiwania wolnych modułów byłaby tak duża, że nie rekompensowało by jej ewentualne poprawienie wydajności.

Interfejs

Interfejs składa się z jednej funkcji „obsługi technicznej” oraz dwóch grup funkcji — do obsługi plików i katalogów.

```
int x_howmanycs(void);
```

Funkcja zwraca informację o ilości uruchomionych lokalnie modułów klienta. Musi być wywołana na początku pracy biblioteki.

Zdefiniowane są dwie funkcje służące do odczytu katalogów i cztery funkcje wprowadzające zmiany w katalogach.

[3] ang. *dæmon*, wywodzące się z łaciny określenie oznaczające działający w tle program usługowy

```
init-+-adjkerntz
     |-cron
     |-dgfsd--+-async
     |        |-10*[clnt]
     |        |-dirmd
     |        |-filemd
     |        |-log
     |        |-tree
     |        `-4*[trans]
     |-dhcpd
     |-8*[getty]
     |-httpd---13*[httpd]
     |-inetd
     |-lpd
     |-master-+-pickup
     |        `-qmgr
     |-mountd
     |-moused
     |-named
     |-nfsd---4*[nfsd]
     |-4*[nfsiod]
     |-nmbd---nmbd
     |-portmap
     |-postgres
     |-rpc.statd
     |-smbd
     |-sshd
     |-syslogd
     `-upsd
```

Rysunek 2.3: Drzewo procesów w systemie FreeBSD Unix z uruchomionym serwerem DGFS

int `x_opendir(x_filename);`

Argument określa jaki katalog ma zostać odczytany. Moduł klienta łączy się z serwerem, który jest właścicielem danego katalogu i pobiera listę plików, którą umieszcza w tymczasowym pliku. Identyfikator ten jest zwracany przez funkcję (dalej biblioteka według otrzymanego identyfikatora odczytuje listę plików i udostępnia klientowi).

int `x_closedir(`*int*`);`

Po wywołaniu tej funkcji moduł usuwa tymczasowy plik według podanego w argumencie identyfikatora.

`int x_erase(x_filename);`
Kasowanie plików — moduł informuje serwer, który jest właścicielem katalogu, w jakim znajduje się dany plik (a dokładnie jego moduł obsługi katalogu) o chęci skasowania pliku.

`int x_rename(x_2filenames);`
Zmiana nazwy pliku — argumentem funkcji jest struktura składające się z dwóch nazw (starej i nowej). Podobnie jak przy kasowaniu moduł klienta informuje odpowiedni moduł katalogu i lokalny moduł obsługi plików o chęci zmiany nazwy a te podejmują odpowiednie działania.

`int x_mkdir(x_filename);`
Tworzenie nowego katalogu. Powoduje wywołanie funkcji `x_makedir` w module obsługi katalogów.

`int x_rmdir(x_filename);`
Usuwanie katalogu. Powoduje wywołanie funkcji `x_removedir` w module obsługi katalogów. Usuwany katalog musi być pusty.

Operacje na plikach sprowadzają się właściwie do dwóch działań — otwarcia pliku i jego zamknięcia, są to jednak dość skomplikowane funkcje. Szczególnie modyfikacje pliku wymagają specyficznych działań. Przykłady zachowań systemu w czasie operacji na plikach przedstawione są w rozdziale 2.2.11.

`int x_openr(x_filename);`
Otwarcie pliku do odczytu. Argumentem funkcji jest nazwa pliku[4]. Moduł klienta dokonuje sprawdzenia (poprzez moduł obsługi plików) czy plik jest dostępny lokalnie. Jeśli nie to dokonuje sprowadzenia pliku z jakiegoś innego serwera. W tym celu kontaktuje się z serwerem-właścicielem katalogu, w którym znajduje się dany plik[5], od którego otrzymuje listę serwerów, na których powinna znajdować się kopia. Wybór jednego z serwerów posiadających kopię może odbywać się w sposób losowy, a w przypdaku problemów z dostępem do wylosowanego serwera następne mogą być wybierane w sposób cykliczny. Następnie moduł dokonuje pobrania kopii z wybranego serwera i umieszcza ją w lokalnych zasobach (oczywiście poprzez moduł obsługi plików), po czym zwraca identyfikator tego pliku (w przypadku gdyby plik był od razu dostępny lokalnie zwrócenie identyfikatora nastąpiłoby natychmiast). O otwarciu pliku jest też informowany moduł obsługi plików.

`int x_openw(x_filename);`
Otwarcie pliku do modyfikacji. Funkcja ta służy do otwarcia pliku w celu jego modyfikacji, a także do stworzenia nowego pliku. Operacja przebiega nieco inaczej niż w przypadku odczytu. Moduł klienta najpierw kontaktuje się z odpowiednim właścicielem katalogu zgłaszając chęć otwarcia pliku do zapisu. Jeśli otrzyma zgodę (może jej nie otrzymać, na przykład z tego powodu, że

[4] wszędzie gdzie używana jest nazwa pliku założone zostało, że chodzi o pełną (bezwzględną) ścieżkę dostępu, jest to konieczne ze względu na brak pojęcia *katalogu bieżącego* na poziomie klienta

[5] odnalezieniem tego serwera zajmuje się opisany dalej moduł drzewa

w tym czasie ktoś inny dokonuje modyfikacji tego pliku) wykonuje operacje analogiczne jak przy odczycie pliku (sprawdzenie lokalnych zasobów i ewentualne pobranie pliku). Różnica występuje w działaniu modułu obsługi plików, moduł klienta otrzymuje od niego identyfikator i tak jak przy otwieraniu do odczytu zwraca go klientowi. W sytuacji gdy plik o podanej nazwie nie istnieje sytuacja prawie się nie zmienia (pomijane zostaje jedynie pobranie pliku).

```
int x_close(int);
```

Zamknięcie pliku — moduł klienta przekazuje otrzymany w argumencie identyfikator dalej do modułu obsługi plików, który wykonuje odpowiednie do posiadanych informacji działania (jego działania zależą od tego czy plik był otwarty do odczytu czy do modyfikacji).

2.2.5. Moduł struktury drzewa

System DGFS zbudowany jest na wspólnym globalnym drzewie katalogów. Oddzielny moduł zajmuje się przeszukiwaniem i zapamiętywaniem struktury tego drzewa. Należy zwrócić uwagę, że nie można za jego pomocą modyfikować struktury katalogowej a jedynie ją przeglądać — do modyfikacją zajmuje się moduł obsługi katalogów.

Interfejs

Interfejs składa się z trzech funkcji. Dwie pierwsze wykonują podobne zadanie.

```
x_dirowner x_getclosestdirowner(x_filename);
```

Funkcja wyszukuje właściciela katalogu podanego jako argument. Jeżeli właściciel nie jest znany to funkcja szuka najbliższego wierzchołka drzewa katalogów takiego, że jego właściciel jest znany (najbliższego w sensie ilości wierzchołków, które trzeba przejść od węzła odpowiadającego podanemu katalogowi, do innego węzła).

```
x_dirowner x_getdirowner(x_filename);
```

Funkcja zwraca właściciela podanego katalogu. Jeżeli go nie zna, to kieruje odpowiednie zapytania do innych serwerów (a konkretnie do modułów drzew tych serwerów) i odnajduje go. W czasie poszukiwania uaktualniane jest pamiętane w module drzewo.

```
struct x_dirowner
       {
       char filename[X_FILENAME_SIZE];
       IP server_addr;
       int is_owner;
       int local;
       };
typedef struct x_dirowner x_dirowner;
```

Obydwie funkcje zwracają strukturę `x_dirowner` określającą właściciela katalogu. Jest ona zbudowana następująco: `filename` jest nazwą katalogu (w przypadku funkcji `x_getclosestdirowner` nie musi to być ten sam katalog co w argumencie funkcji). `server_addr` jest adresem IP właściciela katalogu. Dwie ostatnie składowe są nieco nadmiarowe — `is_owner` mówi czy zwracany właściciel to ten, którego szukaliśmy (można tę informację uzyskać porównując argument funkcji ze składową `filename`), natomiast `local` zawiera wartość *prawda* gdy właściciel katalogu jest właśnie tym serwerem, do którego skierowano pytanie.

```
int x_adddirectorytotree(x_filename);
```

Funkcja służy rozszerzaniu struktury drzewa. Katalog określony przez nazwę zawartą argumencie jest dodawany w odpowiednim miejscu drzewa (o ile to możliwe). Funkcja jest wywoływana zawsze lokalnie przez moduł obsługi katalogów, który będzie obsługiwał tworzony katalog.

```
int x_removedirectoryfromtree(x_filename);
```

Funkcja usuwa dany katalog z drzewa. Wywołujący ją moduł obsługi katalogów dba o to, aby wewnątrz nie znajdowały się żadne pliki lub podkatalogi.

Działanie modułu drzewa określone jest następująco: moduł klienta pytając o jakiś katalog używa funkcji `x_getdirowner`, ponieważ nie interesuje go nic poza adresem właściciela danego katalogu. Inaczej postępuje moduł drzewa chcąc wyszukać informacje — w kontakcie z innymi serwerami używa funkcji `x_getclosestdirowner`, co pozwala mu uzyskać najwięcej informacji o drzewie w trakcie przeszukiwania.

Ważne jest, że informacja o strukturze drzewa jest z założenia niezmienna i po jednokrotnym znalezieniu właściciela moduł nie potrzebuje już go więcej szukać.

Struktury danych

Głównym zadaniem modułu obsługi katalogów jest odwzorowanie nazw katalogów na adresy serwerów obsługujących te katalogi. Informacje przechowywane są w strukturze, która w sposób naturalny odwzorowuje postać drzewa katalogów.

Jest to drzewo o dowolnej liczbie odgałęzień w każdym węźle. Korzeń drzewa odpowiada katalogowi głównemu „/”. Węzły połączone z korzeniem odpowiadają katalogom utworzonym w katalogu głównym. Na przykład jeżeli istnieje ścieżka `/usr/local/src/sys` to węzeł odpowiadający katalogowi `local` jest następnikiem węzła odpowiadającemu katalogowi `usr`, natomiast jednym z jego następników jest węzeł odpowiadający katalogowi `src`.

W każdym węźle znajduje się struktura zawierająca następujące informacje:

- nazwę katalogu
- klucz odpowiadający nazwie (klucz jest wtórny względem nazwy pliku, jedynym jego zastosowaniem jest przyspieszenie przeszukiwania drzewa przez zamianę porównań ciągów tekstowych na porównania liczb)

- adres serwera, który obsługuje dany katalog lub znacznik, że adres serwera nie jest znany
- listę następników danego węzła czyli listę podkatalogów danego katalogu

Taka postać drzewa pozwala określać, który katalog, dla którego znany jest adres serwera, znajduje się „najbliżej" (tak jak jest to zrealizowane w funkcji `x_getclosestdirowner`). Struktury wykorzystane w opisanych niżej modułach obsługi katalogów i obsługi plików są nieco lepsze lecz nie dają takiej możliwości (lepsze znaczy łatwiejsze w implementacji i szybsze w przeszukiwaniu).

Szybkość przeszukiwania drzewa katalogów modułu obsługi drzewa jest proporcjonalna do ilości podkatalogów w analizowanej ścieżce i średniej ilości podkatalogów w katalogu (dla całego drzewa).

2.2.6. Moduł obsługi katalogów

Moduł obsługi katalogów pełni kilka ważnych funkcji w systemie DGFS:

- Obsługa podległego mu wycinka drzewa katalogów.
- Kontrola rozmieszczenia kopii plików.
- Kontrola modyfikacji plików (blokady przy zapisie).
- Kontrola aktualizacji kopii plików.

Interfejs

```
int x_modifyopen(x_filename);
```

Funkcja otwarcia pliku w trybie modyfikacji (służy także do tworzenia nowych plików). Moduł sprawdza czy dany plik nie jest już otwarty w trybie modyfikacji. Jeżeli nie to dokonuje otwarcia (zapamiętując informację o tym, kto i kiedy tego dokonał).

```
int x_modifyclose(x_filename);
```

Funkcja zamykająca plik, który był otwarty do zapisu. Zadaniem modułu jest teraz powiadomienie wszystkich serwerów, które posiadają kopię danego pliku (oprócz tego serwera, na którym dokonano modyfikacji), że ich wersje są nieaktualne.

```
x_fileowners x_getfileowners(x_filename);
```

Funkcja służy do pobierania informacji o rozmieszczeniu kopii plików. Zwracana struktura jest stałej wielkości tablicą zawierającą adresy serwerów, które powinny posiadać kopie plików.)

```
int x_makedir(x_filename);
```

Funkcja tworzy nowy katalog — od tego momentu moduł, w którym została ta funkcja wywołana będzie obsługiwał tenże katalog. Dodatkowo moduł musi o stworzeniu katalogu poinformować moduł obsługujący katalog nadrzędny względem tworzonego katalogu (funkcja `x_adddir`) oraz lokalny moduł drzewa (funkcja `x_adddirtotree`).

`int x_removedir(x_filename);`
Funkcja usuwa dany katalog, o ile nie zawiera on żadnych plików i podkatalogów. Dodatkowo wywoływana jest funkcja `x_removedirfromtree` lokalnego modułu drzewa, który usuwa informacje o tym katalogu. Ważne, że moduły drzew na innych serwerach nadal mogą posiadać informacje o skasowanym właśnie katalogu, które uaktualnią się dopiero po nieudanych próbach dostępu do katalogu.

`int x_adddir(x_filename);`
Funkcja dodaje nowy wpis do odpowiedniego katalogu (dotyczy on stworzonego na innym serwerze podkatalogu, który znajduje się w katalogu obsługiwanym przez dany serwer). Dodatkowo informuje o zdarzeniu lokalny moduł drzewa (funkcja `x_adddirtotree`).

`int x_removefile(x_filename);`
Funkcja usuwa plik. Sprowadza się to do powiadomienia wszystkich posiadaczy kopii o tym, że ich kopia jest nieaktualna — co powoduje skasowanie wszystkich kopii. Do powiadamiania jest używana ta sama funkcja, co przy aktualizacji po modyfikacji pliku (`x_erasefile`).

`int x_renamefile(x_2filenames);`
Funkcja zmienia nazwę pliku. Struktura `x_2filenames` zawiera dwie nazwy — starą i nową. Nie jest możliwe za pomocą tej funkcji przemieszczanie plików między katalogami. Moduł informuje o zmianie nazwy serwery znajdujące się na liście posiadaczy kopii.

`int x_removefromownerslist(x_filename);`
Funkcja usuwa wywołujący serwer z listy przechowujących dany plik, o ile tenże serwer nie jest właścicielem jedynej kopii. Wywoływana przez moduły obsługi plików w czasie opróżniania swoich magazynów (funkcja `x_cleanup`).

Struktury danych

Podobnie jak moduł obsługi drzewa odwzorowywał katalogi na adresy serwerów, moduł obsługi katalogów odwzorowuje nazwy plików na adresy serwerów, które posiadają kopie plików. Najważniejszą różnicą jest to, że serwer obsługi katalogów zna adres przynajmniej jednego serwera dla każdego pliku (ale tylko dla wybranych katalogów).

Do przechowywania danych wykorzystywane jest drzewo binarne, tzw. BST (ang. *Binary Search Tree*). Kluczem uporządkowania drzewa jest pewna wartość obliczana na podstawie nazwy pliku (włącznie z pełną ścieżką).

Każdy węzeł zawiera następujące informacje:

- pełną nazwę pliku
- klucz obliczony na podstawie nazwy
- listę serwerów posiadających kopię tego pliku
- informacje o zablokowaniu pliku do zapisu (kto i kiedy)
- parę wskaźników na następniki tego węzła

Prędkość przeszukiwania drzewa jest proporcjonalna do logarytmu liczby plików we wszystkich katalogach obsługiwanych przez dany serwer.

2.2.7. Moduł obsługi plików

Moduł ten zajmuje się lokalnym magazynem plików. Do jego zadań należy udostępnianie posiadanych plików, utrzymywanie spójności i zapewnianie zachowania odpowiedniej semantyki współdzielenia.

Interfejs

```
int x_havefile(x_filename);
```

Funkcja zwraca informację czy w magazynie znajduje się żądany plik.

```
int x_getfiler(x_filename);
```

Najważniejsza funkcja modułu obsługi plików — zwraca identyfikator pozwalający uzyskać dostęp do pliku o nazwie przekazanej w argumencie. Wywoływana przy otwieraniu pliku do odczytu. Moduł zapamiętuje ile razy była wywoływana ta funkcja tworząc „licznik otwarć" — jest to wartość powiązana z danym plikiem, która określa ilu klientów otworzyło go do odczytu. Wartość licznika jest zwiększana przez funkcję `x_getfiler` a zmniejszana przez `x_ungetfile`.

```
int x_getfilew(x_filename);
```

Funkcja otwarcia pliku do modyfikacji. Działa w następujący sposób — moduł wykonuje kopię pliku o nazwie określonej przez argument i nadaje mu nowy identyfikator (który zostanie zwrócony przez funkcję). Wszystkie późniejsze żądania odczytu tego pliku będą nadal kierowane do starej wersji a dopiero po wykonaniu `x_ungetfile` stara wersja zostanie podmieniona na nową. W przypadku gdy plik o podanej w argumencie nazwie nie istnieje jest to uznane za chęć stworzenia takiego pliku — faktycznie jest tworzony plik o zerowej długości, który otrzymuje odpowiedni identyfikator.

```
int x_ungetfile(int);
```

Odpowiednik funkcji `x_close` modułu klienta. Zmniejsza „licznik otwarć". Gdy licznik otwarć osiągnie zero następuje sprawdzenie, czy w międzyczasie plik nie został zastąpiony nowym lub usunięty — jeżeli coś takiego nastąpiło to moduł dokonuje odpowiednich zmian w swoich strukturach i usuwa go. Taki sposób postępowania wynika z założonej semantyki — klienci, którzy otworzyli plik muszą mieć możliwość dokończenia działań na nim w takim stanie, w jakim był gdy go otwierali. W przypadku awarii klienta polegającej na zakończeniu pracy bez zamknięcia otwartych plików (co oznacza m.in. brak wywołania funkcji `x_ungetfile` i pozostawienie nieprawdziwego stanu licznika otwarć) konieczne jest dokonanie odpowiedniej aktualizacji w asynchronicznej funkcji porządkującej `x_cleanup`.

```
void x_putfile(x_file);
```

Funkcja używana przez moduł klienta w celu umieszczenia pobranej kopii pliku w lokalnym magazynie. Struktura `x_file` zostanie opisana poniżej.

```
struct x_file
        {
```

```
        char filename[X_FILENAME_SIZE];
        int temp_id;
         };
typedef struct x_file x_file;
```

Struktura zawiera dwie informacje — nazwę pliku oraz identyfikator pliku tymczasowego. Po wywołaniu funkcji `x_putfile` moduł znajduje plik według identyfikatora i nadaje mu nowy identyfikator umieszczając go w magazynie plików.

```
void x_erasefile(x_filename);
```

Funkcja usuwająca plik z magazynu, wywoływana przez moduł obsługi katalogów. Moduł obsługi plików sprawdza czy może usunąć plik (tzn. czy jego licznik otwarć jest równy zero). Jeśli nie to zaznacza plik jako przeznaczony do skasowania (jeżeli plik jest zaznaczony „do skasowania" to na żądania dostępu do niego odpowiada: „nie mam takiego pliku"). Warto zauważyć, że nie jest możliwa taka sytuacja, gdy żądanie skasowania będzie dotyczyło pliku właśnie otwartego do zapisu, ponieważ zapisy są kontrolowane przez moduł obsługi katalogów (a ten gdy plik zostanie zmieniony wywołuje funkcję `x_erasefile` u wszystkich posiadaczy kopii, oprócz tego, u którego znajduje się kopia aktualna — właśnie zmieniona).

```
void x_cleanup(void);
```

Funkcja wywoływana przez moduł kontroli asynchronicznej (rozdział 2.2.10). Jej zadaniem jest „oczyszczenie" lokalnego magazynu plików. Oznacza to między innymi skasowanie starych plików tymczasowych i usunięcie starych znaczników otwarcia[6] — jeżeli plik ma niezerowy licznik a jednocześnie był otwierany dawno to może to oznaczać, że klient, który dokonał otwarcia uległ awarii zanim zdążył zamknąć plik. Innym zadaniem funkcji jest usunięcie dawno nie używanych plików i doprowadzenie rozmiaru magazynu do ustalonych rozmiarów. Należy zaznaczyć, że moduł obsługi plików nie wie czy ktoś oprócz niego (tzn. serwer na innej maszynie) posiada dany plik, więc przed jego usunięciem musi prosić o potwierdzenie odpowiedni moduł obsługi katalogu (funkcja `x_removefromownerslist`).

Struktury danych

Moduł obsługi plików dokonuje odwzorowania nazw plików na lokalne identyfikatory, zatem sytuacja wygląda tu podobnie jak w przypadku opisanego wyżej modułu obsługi katalogów. Różnice dotyczą danych zawartych w węzłach drzewa. Tak samo jak w module obsługi katalogów używane jest drzewo BST z kluczem obliczonym na podstawie nazwy.

Każdy węzeł zawiera następujące informacje:

- pełną nazwę pliku
- klucz obliczony na podstawie nazwy
- lokalny identyfikator pliku

[6] „stary" oznacza tu pewien ustalany przez administratora przy starcie systemu parametr; na przykład jeden dzień

- znacznik „przeznaczony do skasowania”

Dodatkowo istnieje drugie drzewo BST uporządkowane według lokalnych identyfikatorów, które służy do przechowywania informacji na temat otwarć danych plików (opisany wyżej „licznik otwarć”), częstości używania itp. Każdy plik istniejący w magazynie musi być zapisany w tym drzewie. Drzewo to jest używane w sytuacji gdy jeden plik logiczny istnieje w kilku wersjach jednocześnie — na przykład w sytuacji gdy jedna jego kopia jest czytana a druga modyfikowana obie mają wpisy w tym drzewie (natomiast istnieje tylko jeden logiczny plik, który jest jednocześnie czytany i pisany, i jemu właśnie odpowiada wpis w pierwszym drzewie).

Prędkość przeszukiwanie obydwu drzew jest w przybliżeniu proporcjonalna do logarytmu ilości plików przechowywanych w magazynie plików.

2.2.8. Moduł transportowy

Powodem powstania modułu transportowego w jego aktualnej postaci była chęć optymalizacji przesyłania danych pomiędzy maszynami. Jest to jedyny moduł, który nie wykorzystuje wywołań RPC jako głównej drogi komunikacji. Zadaniem modułu jest przesyłanie plików ze swojego serwera lokalnego do maszyny, która tego żąda.

Moduł transportowy jest drugim modułem, który może być uruchamiany wielokrotnie (podobnie jak moduł klienta). Jest to podyktowane tym, że transfer plików jest elementem krytycznym systemu i gdyby w danej chwili można było pobierać tylko jeden plik z danego serwera to wolna maszyna pobierająca duży plik zablokowałaby innych klientów na długi czas.

Komunikacja odbywa się tu z wykorzystaniem protokołu TCP. Rozwiązanie takie łatwo uzasadnić pokazując jak wywołanie zdalnych procedur nie jest dostosowane do przesyłania plików:

- Ze względu na maksymalną wielkość argumentu funkcji zdalnej przesłanie dużego pliku wymaga zestawu kolejnych wywołań funkcji. Taka realizacja wymagałaby dodatkowo identyfikacji odbiorców — ponieważ kilku klientów może jednocześnie pobierać różne pliki z tego samego serwera i ich wywołania mogą się przeplatać. W przypadku TCP nawiązywane jest stałe połączenie na czas transferu pliku.
- Każde wywołanie RPC obłożone jest pewnym (niepomijalnym) narzutem czasowym — każdorazowo konieczne jest przejście danych poprzez funkcje XDR[7], otwieranie i zamykanie unixowych deskryptorów gniazdek (gdyż każde zdalne wywołanie funkcji jest oddzielnym połączeniem w sensie TCP/IP).

Procedura transportu wygląda następująco:

- Moduł klienta inicjuje połączenie z modułem transportowym.
- Przekazuje modułowi transportowemu nazwę pliku.
- Moduł transportowy dokonuje „otwarcia” pliku (funkcja `x_getfiler` w module obsługi plików).

[7] standard reprezentacji danych w protokole Sun RPC; rozdział 1.3.4

- Następuje transmisja pliku.
- Po zakończeniu transmisji moduł transportowy „zamyka” plik i przechodzi do oczekiwania na następne połączenia.

2.2.9. Moduł dziennika

Złożoność projektu wymusiła dodanie małego modułu zajmującego się rejestracją występujących w systemie zdarzeń czyli dziennika pracy systemu. Taka rejestracja jest standardowym elementem działania wszelkich systemów, w szczególności unixowych serwerów usług. Poszczególne moduły systemu DGFS są silnie samodzielne lecz tworzą jedną całość — zezwolenie każdemu z nich na tworzenie osobnego dziennika nie miałoby dużego sensu, dlatego istnieje oddzielny moduł, który zbiera informacje od wszystkich pozostałych modułów a następnie przekazuje je dalej. Zasadniczo jedyną drogą, przez którą moduły mogą informować administratora o zaistniałych zdarzeniach jest właśnie moduł dziennika.

Domyślnie moduł dziennika po sformatowaniu informacji otrzymanej od innego modułu (dodaniu znacznika czasu i opisu poziomu ważności komunikatu) wypisuje ją na tzw. standardowe wyjście (w konwencji języka C jest to *stdout*). Można też określić nazwę pliku, do którego mają być zapisywane komunikaty. Jeżeli zaszłaby taka potrzeba można dostosować moduł do komunikacji z unixowym dziennikiem systemowym (*syslog*).

Interfejs

Komunikacja z dziennikiem odbywa się poprzez RPC — zdefiniowana jest tylko jedna funkcja, ale moduły nie wywołują jej bezpośrednio. Istnieje mały interfejs pośredniczący, który pozwala na łatwiejsze korzystanie z dziennika.

```
int log_start(void);
```

Fukncja uruchamia połączenie z modułem dziennika. Wywoływana jest na początku procedury startowej każdego z modułów.

```
enum LOG_LEVEL
     {
     LOG_LEVEL_INFO=0,
     LOG_LEVEL_WARNING,
     LOG_LEVEL_ERROR,
     LOG_LEVEL_FATAL,
     LOG_LEVEL_IMPOSSIBLE
     };
typedef enum LOG_LEVEL LOG_LEVEL;
```

Zostało zdefiniowane pięć rodzajów komunikatów jakie mogą zostać zapisane. W kolejności są to: informacja (wszystkie komunikaty informacyjne), ostrzeżenie (możliwe niebezpieczeństwo ale system funkcjonuje normalnie), błąd (wystąpił błąd, akcja się nie powiodła, system podjął odpowiednie kroki aby naprawić sytuację, np. próbuje ponownie wykonać akcję), poważny błąd (system nie może kontynuować działania w takiej sytuacji), sytuacja niemożliwa

(prawdopodobnie błąd w programie lub w systemie, sytuacja która nie powinna się zdarzyć, np. funkcja zwracająca tylko liczby naturalne zwróciła liczbę ujemną).

```
int log_log(LOG_LEVEL level, char *s);
```

Funkcja wysyła komunikat `s` o poziomie ważności `level`.

```
int log_log_std(LOG_LEVEL level, char *s);
```

Funkcja wysyła komunikat o poziomie ważności `level` który składa się z tekstu `s` i opisu błędu systemowego stworzonego na podstawia zmiennej *errno*.

Interfejs RPC składa się z jednej deklaracji.

```
void x_log(x_logentry);
```

Struktura `x_logentry` zawiera treść komunikatu, jego poziom ważności oraz identyfikator modułu, który ten komunikat wysłał.

2.2.10. Moduł kontroli asynchronicznej

Dodatkowym modułem wspomagającym pracę systemu jest moduł działań asynchronicznych. Nie jest on serwerem w sensie RPC i nie ma możliwości kontaktu z tym modułem. Jego zadaniem jest okresowe wywoływanie pewnych procedur w pozostałych modułach.

Przykładem może tu być operacja czyszczenia modułu obsługi plików: w trakcie dłuższego działania systemu gromadzonych jest wiele plików i raz dziennie moduł kontroli asynchronicznej dokonuje (poprzez odpowiednie wywołanie RPC) „konserwacji” systemu — omówiona wcześniej procedura modułu obsługi plików usuwa pewną ilość plików (kryterium jest tutaj częstość ich używania oraz ilość zajmowanego miejsca w stosunku do ustalonych limitów).

Moduł kontroli asynchronicznej otrzymuje w trakcie startu systemu zestaw danych konfiguracyjnych. Dla każdego zdarzenia ustalane są dwie wartości — przedział czasu pomiędzy dwoma uruchomieniami danej akcji i czas czekania przed ponowieniem wywołania w przypadku gdy nastąpi odmowa wykonania akcji. Każdy moduł ma prawo odmówić wykonania akcji gdy stwierdzi, że w tym momencie jest zbyt obciążony. Ma to istotne znaczenie o tyle, że niektóre ze zdarzeń wywoływanych asynchronicznie mogą być stosunkowo czasochłonne — tak jest na przykład w przypadku czyszczenia modułu obsługi plików.

2.2.11. Przykładowe scenariusze zdarzeń w systemie

Odczyt pliku

Program chce odczytać plik `/usr/bin/emacs`. Zachodzą następujące zdarzenia:

1. Program uruchomiony na maszynie A wywołuje funkcję `dgfs_open` biblioteki *dgfslib* z odpowiednimi parametrami (nazwą pliku i trybem czytania — „`r`”) co powoduje, że biblioteka wywołuje zdalną funkcję `x_openr`

modułu klienta (pominięta została kwestia inicjalizacji biblioteki, połączeń sieciowych itp.).

a) Moduł klienta pyta moduł obsługi plików — czy jesteśmy w posiadaniu pliku `/usr/bin/emacs` (funkcja `x_havefile`). Moduł obsługi plików stwierdza stwierdza, że nie.
b) Pyta moduł drzewa kto jest właścicielem katalogu `/usr/bin/` (funkcja `x_getdirowner`).
   i. Moduł drzewa sprawdza w wewnętrznych strukturach czy nie posiada odpowiedniej informacji — nie.
   ii. Wyszukuje w wewnętrznych strukturach właściciela katalogu najbliższego katalogowi `/usr/bin` i stwierdza, że właścicielem katalogu `/bin` jest maszyna B.
   iii. Wywołuje zdalną procedurę `x_getclosestdirowner` w module drzewa na maszynie B — otrzymuje informację, że właścicielem `/usr/bin` jest maszyna C.
   iv. Uaktualnia swoje informacje zapamiętując maszynę C jako właściciela katalogu `/usr/bin`.
   v. Zwraca modułowi klienta odpowiedź — maszyna C.
c) Moduł klienta wywołuje moduł obsługi katalogów maszyny C pytając położenie pliku `/usr/bin/emacs` — otrzymuje odpowiedź: maszyny D i E. Jednocześnie moduł obsługi katalogów maszyny C na liście powiadomień o aktualizacji zapamiętuje, że maszyna A będzie pobierać ten plik.
d) wybiera (losowo) jako dostawcę pliku maszynę D i łączy się z jej modułem transportowym.
e) dokonuje pobrania pliku z maszyny D i zapisuje go lokalnie
f) informuje moduł obsługi plików (funkcja `x_putfile`), że dany plik został pobrany. Zostaje on skatalogowany przez moduł obsługi plików (otrzymuje odpowiedni nowy identyfikator).
g) wywołuje funkcję otwarcia pliku do odczytu w module obsługi plików, funkcja `x_getfiler`, i otrzymuje lokalny identyfikator).
h) zwraca identyfikator bibliotece.

2. Biblioteka otwiera plik za pomocą funkcji standardowych i zwraca programowi deskryptor.
3. Program (klient) wykonuje potrzebne mu działania korzystając z otrzymanego deskryptora.
4. Program wywołuje funkcję `dgfs_close` biblioteki, a ta zamyka otwarty plik i przekazuje to żądanie dalej do modułu klienta poprzez funkcję RPC `x_close`, moduł klienta wywołuje funkcję zamknięcia w module obsługi plików (`x_ungetfile`).

Modyfikacja pliku

Program chce zmodyfikować plik `/usr/bin/emacs`. Zachodzą następujące zdarzenia:

1. Program uruchomiony na maszynie A wywołuje funkcję `dgfs_open` bi-

blioteki *dgfslib* z odpowiednimi parametrami (nazwą pliku i trybem „w" — pisania) co powoduje, że biblioteka wywołuje funkcję RPC `x_openw` modułu klienta.

a) Moduł klienta pyta moduł obsługi plików — czy jesteśmy w posiadaniu pliku `/usr/bin/emacs`. Moduł stwierdza, że tak. Następnie zwraca lokalny identyfikator pliku włącznie z informacją, kto jest właścicielem katalogu `/usr/bin` (maszyna C).
b) Moduł klienta wywołuje moduł katalogu maszyny C zgłaszając chęć modyfikacji pliku. Moduł katalogu sprawdza czy nikt w tym momencie nie ma tego pliku otwartego do zapisu (czy plik nie jest zablokowany). Ponieważ nie jest, zwraca pozwolenie na modyfikację zapamiętując, kto jej żądał.
c) Moduł klienta wywołuje funkcję otwarcia pliku do zapisu w module obsługi plików — moduł obsługi plików wykonuje kopię pliku, przypisuje tej kopii nowy identyfikator i zwraca go modułowi klienta.
d) Moduł klienta zwraca ten identyfikator bibliotece.

2. Biblioteka otwiera plik za pomocą funkcji standardowych i zwraca programowi deskryptor.
3. Program (klient) wykonuje modyfikację pliku korzystając z otrzymanego deskryptora.
4. Program wywołuje funkcję `dgfs_close` biblioteki, a ta zamyka otwarty plik i przekazuje to żądanie dalej do modułu klienta poprzez funkcję RPC `x_close`
   a) Moduł klienta wywołuje funkcję zamknięcia w module obsługi plików (funkcja `x_ungetfile`).
      i. Moduł obsługi plików przypisuje nowy unikalny identyfikator do pliku `/usr/bin/emacs` tak, aby następne żądania otwarcia odnosiły się do zmodyfikowanego pliku.
      ii. Sprawdza czy możliwe jest usunięcie starej wersji (czy nikt nie ma jej otwartej do odczytu) a skoro tak to usuwa ją.
   b) Moduł klienta informuje moduł obsługi katalogu maszyny C, że zakończono modyfikację pliku.
      i. Moduł katalogu maszyny C uaktualnia informacje w swoich wewnętrznych strukturach (od tej chwili kopię pliku posiada maszyna A).
      ii. Powiadamia (poprzez moduły obsługi plików) wszystkie pozostałe maszyny z listy posiadających plik `/usr/bin/emacs`, że ich kopie są nieaktualne (wtedy kopie zostają usunięte).

2.3. Możliwe rozszerzenia

Po stworzeniu kompletnego projektu systemu DGFS (oraz ewentualnie jego implementacji) następnym etapem jest rozważenie możliwych rozszerzeń. Dopiero analizując projekt jako całość można ocenić, brak jakich mechanizmów może być najbardziej uciążliwy dla jego użytkowników. Poniżej zostaną

przedstawione aspekty, które wymagałyby rozpatrzenia przy tworzeniu następnych wersji systemu DGFS.

Szyfrowanie

System DGFS może obejmować swym zasięgiem sieci rozległe, w związku z czym transmitowane publicznymi kanałami dane narażone są na przechwycenie oraz modyfikacje. Rozwiązaniem jest tu zastosowanie jednego z pakietów do szyfrowania i autentyfikacji danych. Używany w stworzonej wersji protokół zdalnego wywoływania procedur Sun RPC ma możliwość pracy w oparciu o szyfrowanie typu DES (ang. *Data Encryption Standard*) w związku z czym zapewnienie bezpiecznej transmisji w ramach systemu nie powinno być znaczącym problemem.

Prawa dostępu

W przedstawionej wersji systemu wszyscy klienci mają takie same prawa w systemie, co więcej są to pełne prawa. Znaczy to, że każdy klient może zmienić lub usunąć dowolny plik. Takie podejście zakłada odpowiedzialność klientów za swoje działania. W rzeczywistym systemie takie podejście jest bardzo optymistyczne — w przypadku lokalnych systemów unixowych założenie jest odwrotne: każdy klient jest potencjalnie groźny i należy mu dawać minimalne (ale wystarczające do pracy) prawa dostępu do plików. Powiązane jest to z identyfikacją klienta w systemie — klient aby wykonać jakieś działania musi zostać zidentyfikowany przez system, który następnie przydziela mu odpowiednie prawa.

Niektóre z systemów rozproszonych tworzą mechanizmy identyfikacji obejmujące wszystkie systemy składające się na system rozproszony. Zazwyczaj są to rozwiązania wydzielone z systemu plików — niezależne serwery obsługujące tzw. listy dostępowe (ang. *ACL, Access Control List*).

Dodanie podobnych mechanizmów do systemu DGFS wiązałaby się z przebudowaniem wielu składowych systemu. Prawdopodobnie konieczne byłoby dodanie nowego modułu zajmującego się identyfikacją. Należałoby także stworzyć mechanizmy łączące zabezpieczenia systemu DGFS z zabezpieczeniami systemu lokalnego.

Klient zdalny

W stworzonym projekcie istnieje dość silne założenie, że każdemu klientowi odpowiada lokalny względem niego serwer systemu DGFS. Można zastanowić się, czy nie warto dodać możliwości korzystania z systemu przez klientów zdalnych, odpowiadających odłączanym lub działającym okresowo stacjom roboczym (także bezdyskowym o małej pamięci wewnętrznej).

Zmiana nie byłaby trudna do wprowadzenia (wymagałaby ona jedynie zmian w bibliotece użytkownika — być może przygotowania dwóch wersji biblioteki, oraz zmian w module klienta).

Serwer wielowątkowy

W tej chwili serwer systemu składa się z zestawu oddzielnych procesów (modułów), które silnie ze sobą współpracują. Możliwe byłoby połączenie modułów w jeden proces, gdzie każdy moduł byłby jednym wątkiem[8]. Taka zmiana pozwoliła by na silniejszą integrację modułów — między innymi na bezpośrednie korzystanie z danych posiadanych przez inne moduły. Jednak nie ma dotychczas dobrze ustalonych standardów wielowątkowości na poziomie jądra systemu (nawet zbliżone systemy z rodziny BSD prezentują różniące się podejścia). Mogłoby to ograniczyć przenośność systemu DGFS do kilku platform (teraz implementacja jest zasadniczo możliwa na wszystkich systemach unixowych wspierających Sun RPC).

Multimedia

System DGFS nie posiada żadnych mechanizmów pozwalających na obsługę plików jako strumieni a taki właśnie sposób obsługi jest typowy dla dzisiejszych rozwiązań spod znaku multimediów. Strumień danych jest to źródło, które w sensie logicznym nie ma końca — co oznacza, że nie można takiego pliku pobrać w całości. Przykładem takiego strumienia jest transmisja radiowa.

Trudno powiedzieć czy dodawanie obsługi strumieni jest konieczne — faktem jest jednak, że wymagałoby to stworzenia oddzielnych mechanizmów przeznaczonych tylko do tego celu. W chwili obecnej lepiej chyba zdać się na gotowe istniejące rozwiązania zoptymalizowane pod kątem transmisji strumieniowej.

Balansowanie obciążenia

Ciekawym pomysłem jest dodanie do systemu elementów pozwalających na automatyczne równomierne rozkładanie obciążenia pomiędzy serwery (obciążenia w znaczeniu zajętości dysków). Teraz może się bowiem zdarzyć, że próba stworzenia nowego pliku nie powiedzie się z powodu braku miejsca na dysku serwera, przy czym moduł obsługi plików nie będzie mógł usunąć żadnego pliku, gdyż tenże serwer jest posiadaczem jedynych kopii. W systemie z balansowaniem obciążenia w takiej sytuacji nastąpiłoby przesłanie części plików na inny serwer, posiadający wolną przestrzeń dyskową. Oczywiście takie usuwanie musiałoby odbywać się w oparciu o informacje zebrane przez moduł obsługi plików aby nie naruszać filozofii działania systemu — znaczy to, że usuwane (lub przemieszczane) muszą byc pliki, które były najdawniej używane i zarazem o odpowiedniej wielkości (w sytuacji awaryjnej ważne jest aby żądaną ilość miejsca zwolnić szybko, więc nie opłaca się usuwać wielu małych plików).

[8] w dużym uproszczeniu wątek jest to podproces pewnego procesu, który współdzieli pamięć z innymi wątkami tego samego procesu

Spis tabel

1.1 Cztery sposoby postępowania z plikami dzielonymi w systemie rozproszonym 10
1.2 Porównanie wad i zalet serwerów bezstanowych i stanowych 12
1.3 Obserwowane właściwości systemu plików 13

Spis rysunków

1.1 Schemat lokalnego wywołania procedury 20
1.2 Schemat zdalnego wywołania procedury 21

2.1 Schemat połączeń w systemie DGFS 34
2.2 Schemat systemu DGFS w obrębie jednej maszyny 38
2.3 Drzewo procesów w systemie FreeBSD Unix z uruchomionym serwerem DGFS 41

Bibliografia

[1] Andrew S. Tanenbaum: *Rozproszone systemy operacyjne*, Wydawnictwo Naukowe PWN, Warszawa 1997

[2] G. Coulouris, J. Dollimore, T. Kindberg: *Systemy rozproszone, podstawy i projektowanie*, Wydawnictwa Naukowo-Techniczne, Warszawa 1998

[3] W. Richard Stevens: *Programowanie zastosowań sieciowych w systemie Unix*, Wydawnictwa Naukowo-Techniczne, Warszawa 1998

[4] M. Gabassi, B. Dupouy: *Przetwarzanie rozproszone w systemie Unix*, Lupus, Warszawa 1996

[5] A. Silberschatz, P. B. Galvin: *Podstawy systemów operacyjnych*, Wydawnictwa Naukowo-Techniczne, Warszawa 2000

[6] RFC 1057: *RPC: Remote Procedure Call, Protocol Specification, Version 2.*, Sun Microsystems, 1988

[7] RFC 1014: *XDR: External Data Representation Standard*, Sun Microsystems, 1987

[8] Timothy Parker: *TCP/IP*, Helion, Gliwice 1997

[9] Æleen Frisch: *Unix — administracja systemu*, O'Reilly and Associates, Inc. — Wydawnictwo RM, Warszawa 1997