



**WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI**
POLITECHNIKI RZESZOWSKIEJ

Katedra Informatyki i Automatyki

Piotr Maślanka

Rozproszona baza danych
systemu kontrolno-pomiarowego

Praca dyplomowa inżynierska

Opiekun pracy:
dr inż. Andrzej Stec

Rzeszów 2015

Spis treści

1. Wstęp	4
2. Sformułowanie problemu	8
2.1. Ogólne wymagania	8
2.2. Stan bieżący	10
2.2.1. Rozwiązania Big Data	11
2.2.2. Przemysłowe bazy danych typu „historian”	14
2.3. Zagadnienia	15
2.4. Rozproszona tabela mieszająca	19
3. Charakterystyka systemu	22
3.1. System kontrolno-pomiarowy	22
3.2. Przepływ danych	26
3.3. Analiza ilości danych	30
3.4. Założenia i zarys działania	34
3.5. Kluczowe problemy i ich rozwiązanie	39
3.6. Anatomia ciągu	45
4. Opis modułów programu bazodanowego	47
4.1. startup — uruchomienie bazy i konfiguracja	47
4.2. gcollector — odśmiecacz metadanych	47
4.3. gossip — komunikacja stanu węzłów	48
4.4. netdispatch — multiplekser połączeń TCP	51
4.5. lfds — przechowywanie danych ciągu	51
4.6. store — menedżer danych i metadanych ciągów	56
4.7. repair — naprawa uszkodzonych ciągów	61
4.8. ifc — interfejs użytkownika i międzywęzłowy	62
4.9. Protokoły sieciowe	65
5. Instalacja i konfiguracja systemu	71
5.1. Wymagania sprzętowo-programowe	71

5.2. Planowanie klastra	72
5.3. Instalacja i konfiguracja węzła	73
5.4. Naprawa klastra i węzła	78
5.5. Projektowanie i zapis ciągów	79
5.6. Interfejs Python	82
6. Testy systemu	84
6.1. Zależność czasu obliczeń od liczby ciągów	84
6.2. Reakcja systemu na awarię	86
6.3. Test obciążeniowy	89
7. Podsumowanie	92
Spis rysunków	96
Spis tabel	97
Bibliografia	98
Dodatek	99

1. Wstęp

Jedną z podstawowych cech charakteryzujących systemy kontrolno-pomiarowe jest to, że dotyczą one monitorowania procesów zmiennych w czasie. Kontrolowanymi wielkościami mogą być: natężenie prądu, wartość napięcia, temperatura, ciśnienie, zużycie paliwa i wiele innych. Zmiany ich wartości mogą być wywołane zamierzonymi działaniami operatora, ale także różnego rodzaju zakłóceniami lub awariami. Do analizy poprawności funkcjonowania takich systemów konieczne jest często rejestrowanie danych bieżących w dłuższym okresie czasu i przechowywanie ich w bazach danych. Jednocześnie, wraz ze wzrostem liczby oraz złożoności nowo projektowanych systemów, ilość danych niezbędnych do przechowywania stale wzrasta. Istnieje więc potrzeba opracowywania rozwiązań, które będą zapewniać efektywny i niezawodny dostęp do danych, minimalizując jednocześnie ryzyko ich utraty.

Bazy danych typu Big Data stanowią stosunkowo nowy trend w informatyce. Służą one do przechowywania danych strukturalnych o objętości zbyt dużej, by możliwa była ich obróbka tradycyjnymi systemami. Charakteryzują się one rozproszeniem danych na wiele węzłów w klastrze. Dzięki temu zapewniają zwiększoną prędkość odczytu i tolerancję na awarię. Znaczna część istniejących na rynku rozwiązań wewnętrznie bazuje na ideach podejścia Big Data, gdyż na chwilę obecną jedynie ono jest w stanie zapewnić również wymagane właściwości wydajnościowe. Nie tylko one są istotne, gdyż od rozwiązania takiego oczekuje się szeregu specyficznych właściwości. Baza taka musi być w stanie przetworzyć żadaną ilość danych w określonym czasie, zależnym od rozwiązania. Dodatkowymi wymaganiami jest odporność na awarie, zwłaszcza w systemie scentralizowanym, gdzie awaria krytycznego komponentu mogłaby spowodować niedostępność całej usługi.

Podstawowymi jednostkami w bazach systemów kontrolno-pomiarowych są szeregi czasowe¹, często odpowiadające jednemu czujnikowi w systemie

¹ciągi informacji uporządkowane w czasie

automatyki. Zapisują one wartości takiego sensora systemu automatyki w postaci ciągu par stempel czasowy — wartość zmierzona. Stworzone w ten sposób szeregi zawierają dane na temat przebiegu danego procesu technologicznego w czasie. Są więc niezbędne dla analizy poprawności pracy systemu. Można je również wykorzystywać w celu bardziej złożonej analizy (np. korelacyjnej) mającej na celu osiągnięcie założonych przez operatora celów, takich jak optymalizacja pracy systemu, kosztów lub badań nad samym procesem technologicznym. Można też na ich podstawie ustalić, kiedy system ulegnie awarii. Mogą być to nawet proste metody, takie jak sprawdzanie błędu względnego między dwoma wartościami, czy bardziej skomplikowane, jak rachunek całkowity i metody korelacji krzyżowej. Częstość zjawiskiem jest stosowanie złożonych analiz, obciążających bazę pod względem ilości odczytów. Musi ona więc pozwalać zarówno na obsłużenie rejestracji dużej ilości danych, jak i na późniejszy dostęp do nich. Dlatego właśnie dobra baza danych przechowująca dane generowane przez system pomiarowy jest kluczowym składnikiem wielu systemów automatyki.

Celem pracy jest opracowanie rozwiązania bazodanowego umożliwiającego gromadzenie danych pomiarowych pochodzących z wielu niedużych systemów automatyki, takich jak kotłownie czy węzły cieplne, w sposób zapewniający ciągłość i niezawodność pracy systemu. Jednocześnie, dane zgromadzone w bazie zapisywane powinny być sposobem pozwalającym na ich swobodny odczyt, umożliwiając operatorowi systemu przeprowadzanie analiz i sporządzanie wykresów.

Rozwiązanie bazodanowe musi być w stanie sprostać wymaganiom dużego rozproszonego systemu kontrolno-pomiarowego do nadzorowania znacznej ilości oddzielonych systemów ogrzewania takich jak kotłownie czy węzły cieplne. Przede wszystkim musi być w stanie obsłużyć obciążenie opisane w p. 3.3. Dodatkowo, musi ono być odporne na awarię pojedynczego węzła, nie wpływając negatywnie na poprawność i istotną jakość działania systemu. Warunkuje to jego rozproszony charakter. Wymagana jest również, ze względu na stały rozwój systemu, możliwość modyfikowania tego rozwiązania w późniejszym czasie.

Opracowywane rozwiązanie jest przygotowywane z myślą o jego praktycznym wykorzystaniu. Autor niniejszej pracy zamierza je w przyszłości zastosować do gromadzenia danych w pracującym już systemie kontrolno-pomiarowym SMOK² [2] służącym do nadzorowania wielu odległych instalacji techniki grzewczej w całej Polsce. Operatorem systemu jest jedna z rzeszowskich firm.

System SMOK jest rozwiązaniem, które cały czas dynamicznie się rozwija. Ze względu na chęć sprawowania kontroli nad własnością intelektualną, również baza danych stanowiąca jego istotny element, została zaplanowana jako autorskie rozwiązanie. Baza danych, która ma stanowić krytyczny komponent, musi również być w stanie sprostać temu rozwojowi. Przewiduje się, że także ona będzie w przyszłości dostosowywana do zmieniających się potrzeb systemu. Pozostałe, techniczne własności rozwiązania zostały ustalone w drodze analizy wymagań.

W ramach pracy wykonano następujące zadania cząstkowe:

- przeprowadzono analizę wymagań systemu dotyczących obciążenia, prędkości pozyskiwania danych i ustalenia wolumenu danych, które system będzie przetwarzał,
- zapoznano się z istniejącymi podobnymi opracowaniami oraz metodami rozwiązywania zasadniczych problemów występujących w tym zagadnieniu,
- opracowano i zaprogramowano od postaw rozwiązanie, które może zapewniać żadaną funkcjonalność,
- przeprowadzono testy weryfikujące stosowalność bazy w systemie SMOK oraz określające charakterystykę jego skalowalności,
- sprzężono rozwiązanie z testową instalacją systemu SMOK.

Omawiane w pracy zagadnienia zostały podzielone na pięć części, z czego pierwsza dotyczy ogólnego sformułowania problemu, a pozostałe poświęcone są opisowi opracowanego rozwiązania.

²System Monitoringu Odległych Kotłowni

W rozdziale 2 opisano ogólnie problem tworzenia rozproszonej bazy danych systemu kontrolno-pomiarowego, wymagania stawiane takim rozwiązaniom oraz istniejące rozwiązania. Opisano również zagadnienia rozproszenia, skalowalności i wyboru technologii za pomocą której można zbudować taką bazę.

W rozdziale 3 scharakteryzowano system kontrolno-pomiarowy, dla którego budowane jest zaproponowane rozwiązanie, przedstawiono wymagania oraz planowane do uzyskania efekty. Dodatkowo zawiera on opis architektury rozwiązania oraz sposób poradzenia sobie z kluczowymi problemami.

Rozdział 4 zawiera szczegółowy opis programu bazodanowego. Przedstawiono w nim dokładne zasady działania poszczególnych modułów systemu, omówiono także najważniejsze fragmenty kodu źródłowego.

W rozdziale 5 zamieszczono ogólne wytyczne dotyczące instalacji oraz konfiguracji. Opisano tam także w jaki sposób projektować klaster bazy, jej schemat oraz instrukcje naprawy w przypadku awarii. Rozdział kończy się opisem interfejsu do bazy danych napisanego w języku Python.

Rozdział 6 zawiera specyfikację, wyniki i wnioski z przeprowadzonych testów. Testy te dotyczą charakterystyki stworzonego rozwiązania, oraz weryfikację spełniania wymogów systemu SMOK.

2. Sformułowanie problemu

W rozdziale opisano ogólnie wymagania, które projektowane rozwiązanie musi spełnić. Opisano również stan bieżący, to jest pokazano obecne sposoby podejścia do rozwiązywania podobnych problemów. Przedstawiono trzy kluczowe zagadnienia dotyczące sposobu rozproszenia systemu oraz wyboru języka programowania. Na końcu rozdziału wspomniano o algorytmie pozwalającym odnajdywać dane w dużych systemach rozproszonych.

2.1. Ogólne wymagania

Przede wszystkim podstawowym wymaganiem projektowanego systemu jest konieczność przechowywania dużej ilości danych. Prognozę³ wzrostu ilości zgromadzonych danych, celem zobrazowania skali, dla pewnych rozmiarów rozproszonych systemów sterowania przedstawiono w tabeli 1⁴. Do danych tych dostęp uzyskuje się stosunkowo rzadko, tak więc rozwiązanie może wykorzystywać kompresję. Ze względu na płynne zmiany między pomiarami spodziewać można się, że dobry efekt przyniosłaby tutaj kompresja typu delta. Polega ona na kodowaniu jedynie zmian między wartościami, do zapisania których wymagane jest zazwyczaj mniej bitów niż do zapisania całej nowej wartości.

Drugim zagadnieniem jest charakter zapisu danych do takiej bazy. Odbywają się one okresowo (częstotliwość zapisów jest bardzo stabilna). Objętość zapisywanych danych nie jest duża (ok. 1,6 MB na minutę dla systemu dużego z tab. 1), jednak równomiernie rozłożona między wszystkie ciągi. Zapisy takie nie mogą zbyt długo oczekiwać na realizację, albowiem podsystem pomiarowy ciągle generuje nowe dane. Nowe rekordy

³Założono 4 bajty na wartość, 8 bajtów na stempel czasowy. Założono że pomiar wykonuje się co 5 sekund. W prognozie pominięto metadane, gdyż stanowią pomijalny udział.

⁴Obliczono je mnożąc 12 bajtów (długość całego rekordu wraz z stemplem) razy 12 (liczba pomiarów na minutę) razy 1440 (liczba minut w dobie) razy 730 (liczba dni w dwóch latach) razy liczbę punktów pomiarowych

Tabela 1: Minimalna ilość miejsca na niekompresowanych danych przechowywanych przez 2 lata przy przykładowych rozmiarach systemów kontrolno-pomiarowych.

Rozmiar	Liczba punktów pomiarowych	Ilość danych
Mały	800	121,1 GB
Średni	4000	605,5 GB
Duży	12000	1817 GB

pojawiają się w czasie rzeczywistym i zbierane są z rozproszonych sensorów — obecnych w całej kontrolowanej instalacji. Rozmiar rekordu danych które zbiera konkretny sensor jest stały. Mogą być to wartości fizyczne, skalarne bądź wektorowe, jednak po kwantyzacji mają zazwyczaj taki sam rozmiar. Dlatego założeniem bezpiecznym jest że rozmiar rekordu jest stały. Dodatkowo, zakładamy że zapisy te są ważne, to znaczy w momencie gdy baza przyjmie żądanie, musi istnieć pewność że dane rzeczywiście zostały zapisane i awaria nie spowoduje ich utraty.

Kolejne wymaganie dotyczy odczytu danych. Dla takiej bazy zachodzi prawidłowość, że im starsze dane tym dostęp do nich jest realizowany rzadziej. Najczęściej wymagany jest dostęp do danych bieżących oraz krótkookresowych. Wyświetla się je na ekranach synoptycznych monitorujących stan systemu, a także wykorzystuje do analizy jego dotychczasowego działania lub przewidywania możliwych do wystąpienia zagrożeń. Dane starsze są wykorzystywane dużo rzadziej, najczęściej do celów tzw. głębokiej analizy. Zapytania dotyczą danych archiwalnych „od do”, konieczne jest więc stosowanie stempli czasowych umożliwiających wyszukanie informacji z żądanych okresów.

Bardzo ważnym aspektem jest także zapewnienie tolerancji na awarie. Większość elementów systemu kontrolno-pomiarowego wspierającego zadania krytyczne wyposażonych jest w komponenty nadmiarowe (redundantne) umożliwiające natychmiastowe zastąpienie uszkodzonego komponentu i przejęcie realizowanej przez niego funkcji. Mogą to być zapasowe interfejsy komunikacyjne i zasilacze, czy nawet procesory i pamięci. Nie

inaczej musi być w przypadku systemu bazodanowego. Odporność na awarie i redundancja jest podstawowym powodem rozproszenia bazy danych, czyli umieszczenia jej na kilku fizycznych maszynach jednocześnie. Mechanizm taki jest w stanie zapewnić ciągłość pracy systemu w przypadku awarii jednego z urządzeń, a przy korzystaniu z wielu lokalizacji także przy czasowej utracie połączenia z którymkolwiek z węzłów sieci.

Ciekawym rozwiązaniem, postulowanym przez General Electric [8] staje się umieszczenie danych przemysłowych w chmurze. Polega to na umieszczaniu tychże danych na serwerach dostawcy oprogramowania, którego zadaniem jest utrzymywanie infrastruktury bazodanowej oraz dbaniu o ich bezpieczeństwo. Z punktu widzenia użytkownika przestaje wtedy być istotna dokładna architektura systemów dostawcy, gdyż nie stanowi ona odpowiedzialności klienta. Rozwiązanie to bywa korzystne, gdyż zwalnia użytkownika z obowiązku zakupu i utrzymywania systemów bazodanowych. Rodzi jednak pewnie wątpliwości co do poufności i bezpieczeństwa tych danych. Znajdują się one bowiem pod kontrolą właściciela „chmury”. Bezpieczeństwa danych chronią wtedy jedynie umowy (np. o zachowaniu poufności). Nie chronią one jednak przed złośliwym wyciekiem danych czy włamaniem na serwery dostawcy. Każdorazowo przed zdecydowaniem się na takie rozwiązanie należy więc przeprowadzić analizę ryzyka.

2.2. Stan bieżący

Rozwiązywany problem nie jest nowy. Istnieje szereg rozwiązań problemu bazy danych systemu kontrolno-pomiarowego przechowujących dane o których wspomniano. Wyróżnić można trzy rodzaje rozwiązań tego typu:

- rozwiązania oparte na relacyjnych bazach danych,
- rozwiązania typu Big Data,
- przemysłowe bazy danych typu „historian”.

Rozwiązania oparte na relacyjnych bazach danych mają zastosowanie tylko w małych systemach pomiarowych. Ze względu na szybki wzrost

indeksów przy składowaniu większych ilości danych przechowywanie i odzyskiwanie danych staje się powolne. Mają natomiast znaczące zastosowanie przy obróbce ciągów stosowanej do przewidywania pewnych zachowań w przyszłości i głębokiej analizy⁵ — zwłaszcza w planowaniu biznesowym — czego dowodzi obecność na rynku takich rozwiązań jak *Microsoft Time Series Algorithm* czy *Oracle8 Time Series Cartridge*. Są to rozszerzenia dodające funkcje obsługi szeregów czasowych w wydajny sposób do istniejących relacyjnych baz danych (w tym wypadku odpowiednio MSSQL Server i Oracle Database).

2.2.1. Rozwiązania Big Data

Rozwiązania Big Data⁶ pojawiły się przy rozwiązywaniu innego, choć pokrewnego problemu. Mierzenie stanu „zdrowia” dużych systemów informatycznych dokonuje się bardzo podobnie jak w przypadku instalacji automatyki, mianowicie określić można serię skalnych wartości opisujących pracę systemu. Tak jak w systemach automatyki, muszą one mieścić się w pewnych „widełkach”. Również do analizy ich pracy stosuje się metody analogiczne jak w automatyce. Nie dziwi zatem, że rozwiązania które powstały do celów obsługi takich systemów są podobne. Niewiele stoi na przeszkodzie by tego typu systemu z powodzeniem wykorzystywać w systemach automatyki, gdzie będą miały przewagę nad dużą częścią rozwiązań „przemysłowych”, zwłaszcza tworzonych przez mniej zaawansowane technicznie firmy.

Do monitorowania małych systemów wystarczające są proste narzędzia pokroju rrdtool [12], które opierają się na lokalnych (realizowanych jako pliki) buforach cyklicznych⁷ monitorujących jakiś okres wstecz pracy danego systemu. Ze względu na brak redundancji, uproszczoną obsługę,

⁵czyli stosowanie zaawansowanych algorytmów statystycznych w celu odkrycia nowych korelacji i powiązań między zebranymi danymi

⁶termin służący do opisu zbiorów danych zbyt dużych by można było je przetwarzać tradycyjnymi metodami oraz rozwiązań służących do przetwarzania tychże.

⁷Bufory takie posiadają stałą długość, a dopisywanie danych gdy bufor jest pełny polega na nadpisaniu „starych” danych

brak wsparcia dla sieciowości i niewielką rozszerzalność (formuła bufora cyklicznego) nie stanowią „uniwersalnej” formuły.

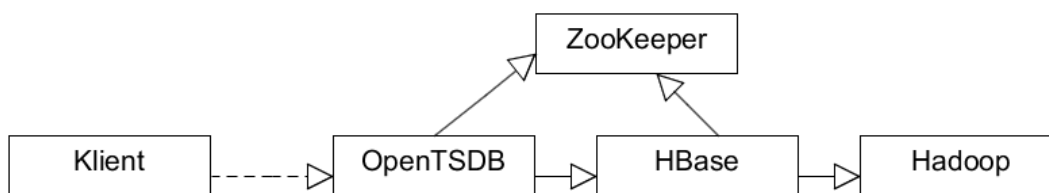
Stosowane rozwiązania tego typu, które mogą mieć zastosowanie w przemyśle, są rozwiązaniami opartymi na schemacie „Big Data”. Stanowią one często rozszerzenia do bardzo elastycznych systemów bazodanowych, zdolnych przechowywać setki terabajtów danych oraz obsługiwać znaczne przepustowości zapisów i odczytów.

Przykładem typowej bazy Big Data może być, stworzona na wzór rozwiązań firmy Google, HBase [5]. Jest to rozproszona i nierelacyjna baza danych, przypisującą kluczowi (będącemu ciągiem znaków) ciąg bajtów. Praktyka wykazała, że na tym paradygmacie da się zbudować inne sposoby przechowywania danych. HBase wspierane przez rozproszony system plików Hadoop [14], który stanowi bazę zapewniającą obsługę dużych zbiorów danych oraz bezpieczeństwo poprzez nadmiarowość.

Na paradygmacie przechowywania danych HBase da się również zbudować rozwiązania służące do przechowywania danych o innym charakterze. Jednym z nich jest, służąca do przechowywania szeregów czasowych, OpenTSDB. Charakteryzuje ją bardzo dobra skalowalność (strona internetowa tego projektu wspomina o milionach zapisów na sekundę) oraz charakterystyczna dla rozwiązań Big Data nadmiarowość, wspierana przez HBase.

Ogólną architekturę rozwiązania opartego o OpenTSDB przedstawiono na rysunku 1. Dodatkowym, wymaganym elementem jest system monitoringu i komunikacji międzywęzłowej ZooKeeper. Mimo bardzo dobrych właściwości wdrożenie OpenTSDB pozostaje istotnym problemem i wymaga wysoko przeszkolonej kadry. Głównym powodem jest mnogość rozwiązań wymagających instalacji i konfiguracji przed uruchomieniem OpenTSDB.

Rozwiązanie to jest głównie zorientowane na wizualizację danych, dysponując zintegrowanymi rozwiązaniami pozwalającymi na „wygładzanie” brakujących punktów danych. Czasami dokonuje ono manipulacji danymi w celu skutecznego przedstawiania danych w formie graficznej. OpenTSDB na prezentacji danych kładzie nacisk do tego stopnia, że jednym z wyma-



Rysunek 1: Architektura rozwiązania OpenTSDB. Klient jest elementem zapisującym dane lub odczytującym je. Strzałki obrazują relację „wymaga”.

Tabela 2: Zestawienie wspomnianych rozwiązań opartych na Big Data

Rozwiązanie	Podstawowa baza danych
KairosDB	Cassandra
OpenTSDB	HBase

gań przy instalacji jest sprawny pakiet do rysowania wykresów *gnuplot*.

Podobnym rozwiązaniem do OpenTSDB jest projekt, który wyrósł na bazie tegoż, czyli KairosDB. Jest to baza dużo bardziej zorientowana na samo przechowywanie danych niż OpenTSDB, jednak jej zasadniczą cechą jest możliwość korzystania zarówno z HBase, jak i z zalecanego zdecentralizowanego rozwiązania Big Data pod nazwą Cassandra [7] (reprezentujący identyczny paradygmat bazodanowy jak HBase). W przeciwieństwie do poprzedniego, ze względu na bazę na której się opiera (tj. Cassandra) KairosDB nie ma pojedynczego punktu awarii jakim jest węzeł typu „master” w HBase, ma także lepsze charakterystyki prędkości i skalowalności [11]. Widać tutaj, że oba rozwiązania są do siebie podobne, reprezentując naskładki na odpowiednie bazy. Zestawienie to przedstawiono w tabeli 2.

Głównym parametrem przemawiającym za wybraniem któregośkolwiek rozwiązania jest posiadanie funkcjonującego klastra jednego z dwóch systemów. Konfiguracja klastra, zarówno HBase jak i Cassandra, jest bowiem nietrywialna⁸. Zastosowania obu tych baz są tak szerokie, że zwykle przechowywanie szeregów czasowych jest tylko częścią zgromadzonych tam danych.

Pewnym problemem tych baz jest charakter przyjmowania przez nie

⁸Choć klaster Cassandra wymaga rząd wielkości mniej czasu do skonfigurowania.

zadań. Ze względu na to że zostały stworzone w celu zbierania metryk zachowania systemu, a nie pomiarów, nie czynią większych gwarancji na to, że zebrane dane przetrwają. Możliwy jest scenariusz w którym baza otrzyma rozkaz zapisu, potwierdzi go, a awaria następująca 5 sekund potem spowoduje trwałą utratę tych danych. Bierze się to z agresywnego przetrzymywania w pamięci danych, by można było później jednorazowo wpisać większą ilość danych, co zwiększa prędkość działania systemu.

2.2.2. Przemysłowe bazy danych typu „historian“

Osobną kategorią są rozwiązania tworzone typowo pod problem przechowywania danych pochodzących z pomiarowych systemów na instalacjach automatyki. Znane są one pod nazwą „operational historian“ bądź „enterprise historian“. Różnicą jest przeznaczenie. „Operational historian“ ma docelowo służyć inżynierom monitorującym procesy technologiczne, zaś „enterprise historian“ ma zbierać dane do celów procesów biznesowych. Ze względu na coraz większą konwergencję rozwiązań obserwuje się zanik rozróżnień (poza celami marketingowymi). Znaczna większość z nich pozwala na zbieranie danych za pomocą interfejsu OPC HDA [10]. Rozwiązanie to zostało zbudowane na bazie Microsoft OLE, i jako część standardu OPC stanowi przemysłowy standard w branży SCADA⁹. Tego typu rozwiązania logują również, dla każdego rekordu, wartość „jakości próbki“, charakterystyczną w zastosowaniach przemysłowych, gdzie dane mogą być chwilowo niedostępne (np. awaria czujnika) bądź gorszej jakości (inne problemy z czujnikiem). W odróżnieniu od baz typu Big Data, każdy pomiar traktowany jest tutaj jako integralna część szeregu, a utrata większej ilości danych w przypadku awarii jest mniej prawdopodobna.

Dane na temat zasad działania tego typu systemów są wyjątkowo trudno osiągalne. Nie budzi to zdziwienia, ze względu na to, że opracowanie takich systemów wymaga dużych nakładów ze strony firm, a upublicznianie zastosowanych technologii mogłoby podkopać przewagę

⁹czyli systemów zapewniających nadzór i pomiarowanie dla procesów technologicznych

biznesową firmy. Typową cechą, odróżniającą te rozwiązania od typu Big Data, jest stosowanie wyspecjalizowanych rozwiązań mających na celu ochronę danych przed utratą w razie awarii. Zwykle jest to konfigurowanie systemu do pracy z dużymi redundantnymi macierzami dyskowymi, zamiast korzystanie z wielu tanich komputerów.

Przykładem może być tutaj Proficy Historian firmy GE [4]. Rozwiązanie to pozwala dodatkowo na eksport danych do systemów Big Data (Hadoop). Innym rozwiązaniem jest Uniformance PHD [6] firmy Honeywell. Podobnym rozwiązaniem, jednak zorientowanym bardziej na pomiary laboratoryjne jest Citadel [9] firmy National Instruments. Mimo bliskiej integracji z środowiskiem LabView dysponuje ono interfejsem OPC.

Interesującym trendem w przemyśle jest umieszczanie historycznych danych procesowych w chmurze [8]. Jedno z rozszerzeń Proficy Historian umożliwia transfer danych na serwery GE, gdzie mogą być one poddane algorytmom przewidującym czy po prostu składowane.

W ogólności, obserwować można konwergencję rozwiązań „historian” i Big Data. Z odpowiednimi dostosowaniami mogą stanowić rozszerzenia wymienne. Dużą zaletą Big Data jest zdolność do przechowywania również innego charakteru danych i bardzo dobre charakterystyki skalowalności, natomiast główną zaletą rozwiązań „historian” jest obsługa standardów OPC i (czasami) wbudowane algorytmy analityczne. Wewnętrznie bazy „historian” często bardzo przypominają rozwiązania Big Data, co ze względu na restrykcyjne licencje, chroniące rozwiązania przed analizą, ciężko zaobserwować.

2.3. Zagadnienia

Podstawowym problemem baz tego typu jest rozproszenie — czy to w celu odporności na awarie, czy zwiększenia wydajności. Głównym zagadnieniem jest tutaj koordynacja. Węzły systemu muszą wiedzieć za jakie partie danych odpowiadają, oraz mieć określony sposób reagowania na komendy. W zasadzie wyróżnić tutaj można dwa podejścia

- Z węzłem/węzłami typu „master”. Jeden węzeł koordynuje pracę

pozostałych. W istotny sposób upraszcza to pracę systemu i zwiększa jego szybkość działania, za cenę wprowadzenia jednak pojedynczego punktu awarii. Problem z jego działaniem w dramatyczny sposób wpływa na działanie całego klastra. Problemy te można do pewnego stopnia zażegnać poprzez wprowadzanie zapasowych węzłów „master“, jednak nie eliminują one tego problemu zupełnie. Przykładem jest system HBase.

- Bez węzła typu „master“. Wszystkie węzły są równorzędne, tak więc eliminuje się pojedynczy punkt awarii. W tym wypadku wymagane są jednak dość złożone algorytmy współpracy i koordynacji aby kłaster mógł pracować. Przykładem jest system Cassandra.

Typowe dla systemów o wysokiej niezawodności jest podejście bez węzła „master“. Często jednak przewaga prostoty systemu z „masterem“ wygrywa, zwłaszcza w systemach, w których analiza ryzyka wykazała że nie jest niezbędna dość wysoka niezawodność. Kolejnym zagadnieniem jest skalowalność. Istnieją dwa podstawowe podejścia do skalowalności, które oczywiście często się ze sobą łączą. Są to:

- Skalowalność pionowa (ang. *scale up*). Tyczy się ona rozszerzania możliwości pojedynczego węzła systemu (większe pamięci, mocniejsze procesory) w celu przetworzenia większych obciążeń. Typowym przykładem takiej skalowalności są systemy typu mainframe. Są to systemy charakteryzujące się dużą wewnętrzną nadmiarowością, dzięki czemu mają wysoką dostępność i odporność na błędy. Dysponują funkcjonalnością która pozwala na rozszerzanie ich w dużym stopniu.
- Skalowalność pozioma (ang. *scale out*). Tyczy się ona dodawania kolejnych węzłów do systemu w celu przetworzenia większych obciążeń. Typowym przykładem takiej skalowalności są bazy danych stosowane przez Google czy Facebook - tysiące nieskomplikowanych sprzętowo węzłów typu ARM bądź Intel.

W zasadzie systemy przetwarzające bardzo duże ilości danych preferują skalowalność poziomą. Istnieją skuteczne algorytmy pozwalające na taki podział danych, aby awaria pojedynczego węzła nie wpłynęła destruktywnie na działanie klastra. Dodatkowo, obecna technologia nie dysponuje możliwością stworzenia pojedynczego serwera który mógłby przetwarzać cały zbiór danych. W niektórych przypadkach udaje się przetwarzać duże zasoby danych na pojedynczych węzłach (np. mainframe), jednak ta tematyka wykracza poza zakres tej pracy, chociażby ze względu na wątpliwą stosowalność rozwiązań mainframeowych w przechowywaniu szeregów czasowych. Skalowalność pionowa ma także dużo gorsze parametry niezawodnościowe, gdyż awaria systemu powoduje jego zupełną niedostępność. Wyjątkiem są tutaj systemy redundatne (takie jak mainframe bądź wybrane systemy typu Intel), jednak koszt takich technologii jest znacznie wyższy niż analogiczne przy skalowalności poziomej.

Wymienione tutaj zagadnienia — charakter rozproszenia i skalowalność — mają istotne znaczenie w projektowaniu systemu informatycznego, którego zadaniem jest przetwarzanie i przechowywanie dużych ilości danych. Decydują one zarówno o możliwościach (ilościowych) obróbki tychże, a również o charakterystyce niezawodnościowej.

Istotnym problemem jest również wybór języka programowania. Stanowi on kompromis między prędkością (w ujęciu czasowym) pisania kodu, prostotą jego późniejszego utrzymania i szybkością rozwiązania stworzonego przy użyciu takiego języka. Jeśli chodzi o tworzenie aplikacji serwerowych i bazodanowych, stosuje się języki, które pozwalają na otrzymywanie szybko działających programów, takie jak C, C++, Java i C# (oraz pokrewne). Rozwój kompilatorów JIT¹⁰. pozwolił na znaczne zminimalizowanie różnic między C/C++ oraz Java/C#. W tym momencie istotne stają się także koszty stworzenia takiego rozwiązania oraz późniejszego utrzymywania go. Funkcje takie jak odśmiecacz (ang. *garbage collector*¹¹)

¹⁰Kompilator JIT pozwala przyspieszyć działanie kodu interpretowanego (stosowanego w Java i C#) poprzez kompilację ich do kodu maszynowego w trakcie działania programu

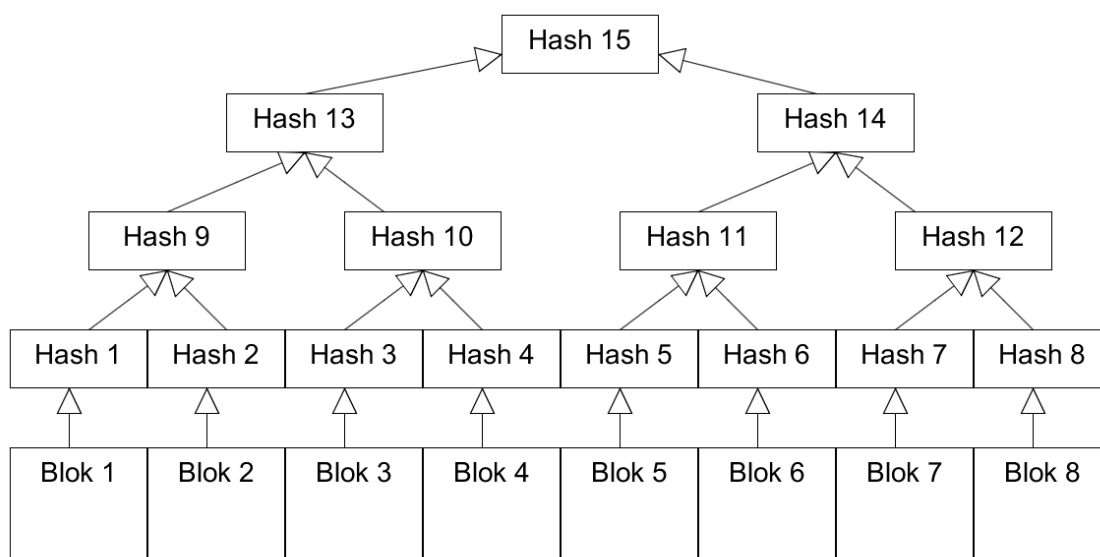
¹¹Moduł który automatycznie determinuje czy dana zmienna jest wykorzystywana, a jeśli nie — zwalnijący zajmowaną przez nią pamięć

obecne w nowoczesnych językach programowania pozwalają na przyspieszenie pracy programisty i minimalizację błędów. Wymienione wyżej języki są najczęściej stosowane do pisania oprogramowania bazodanowego (z drobnymi problemami natury przenośności w przypadku C#).

Ostatnim już omówionym w tej sekcji zagadnieniem jest replikacja danych. Pod tym pojęciem kryje się przechowywanie identycznych kopii danych na niezależnych węzłach. Jeśli jeden z nich ulegnie awarii, dane wciąż pozostają dostępne w klastrze. Określenie, które węzły w systemie rozproszonym odpowiadają za dane porcje danych, a następnie upewnienie się, że wszystkie dysponują identycznymi zestawami, jest zadaniem złożonym. Jeśli dysponujemy systemem z węzłem „master“, to nie stanowi większego problemu rozdzielenie obciążenia między węzły. „Master“ może z powodzeniem dokonywać alokacji i informować poszczególne węzły o ich zakresie odpowiedzialności. Problem pojawia się w systemach zdecentralizowanych, gdzie węzły muszą ustalić ten zakres. Ze względu na łatwość użycia i implementacji często stosuje się w tej sytuacji rozwiązania oparte na funkcjach mieszających, z których jedno opisano w p. 2.4.

Większość rozwiązań wykrywających uszkodzenia danych ciągu i decydujących które węzły dysponują danymi właściwymi opiera się na różnego rodzaju sumach kontrolnych. Jednym z rozwiązań są drzewa Merkle'a, w których dane dzieli się na bloki, z których oblicza się sumę kontrolną. Następnie z kilku tych wartości liczona jest kolejna, itd. Obliczenia mogą być wykonywane równolegle ze względu na drzewiasty charakter, dzięki czemu znajdują częste zastosowanie w systemach bazodanowych tego typu. Ideę algorytmu ilustruje rysunek 2. Metoda ta umożliwia szybkie wyliczenie sumy kontrolnej dla danych, a równocześnie, w razie wystąpienia różnic, określenia której części danych dotyczy problem (porównanie wartości sum kontrolnych „niższego poziomu”).

W przypadku danych szeregów czasowych, gdzie zazwyczaj dane zapisane w blokach „przeszłości“ nie będą już modyfikowane, przypadek drzew Merkle'a można zdegenerować do łańcucha liczącego sumę kontrolną na podstawie własnych danych i sumy kontrolnej poprzedniego węzła. Po-



Rysunek 2: Ilustracja metody drzew Merkle'a

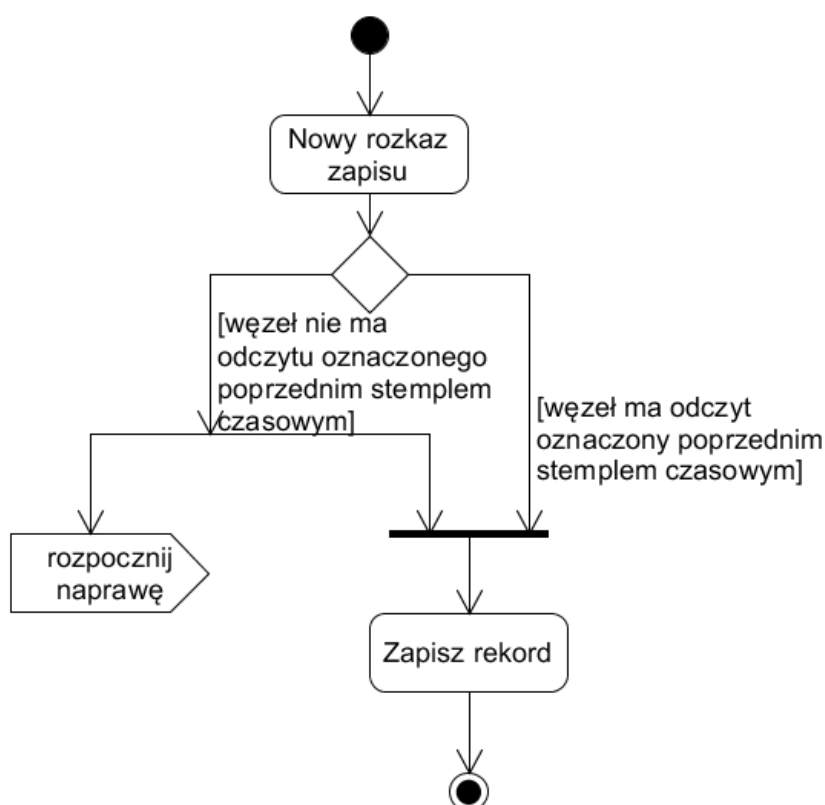
dejsie to jednak nie jest do zastosowania w systemach umożliwiającym modyfikację przeszłych danych, chociażby takich jak zgodne z OPC HDA.

Możliwe są oczywiście inne metody sprawdzania spójności ciągu. W projektowanym nowym rozwiązaniu zdecydowano się na nowatorską metodę. Polega ona na wymaganiu podania, przy poleceniu dopisania nowego rekordu, stempla czasowego poprzedniego rekordu. Element zapisujący dysponuje tą informacją, zaś węzeł na tej podstawie może stwierdzić czy zarejestrował cały przebieg, czy niektórych elementów mu brakuje. Tak czy inaczej, powinien przyjąć zapis - pozostałe przecież nie mogą czekać. Schemat blokowy ilustrujący tą ideę przedstawiono na rysunku 3.

2.4. Rozproszona tabela mieszająca

Algorytm rozproszonej tabeli mieszającej (ang. DHT) jest popularnym sposobem przyporządkowania danych do konkretnych węzłów oraz odzukiwania tych informacji, stosowanych w rozproszonych systemach bazodanowych.

Fundamentalnie zakłada on abstrakcję tablicy asocjacyjnej, czyli zbioru dwuelementowych krotek (*klucz, wartość*), gdzie klucz najczęściej będzie ciągiem znaków. Aby zastosować teraz tabelę mieszającą, wymagane jest obliczenie dodatkowo funkcji mieszającej. Funkcja taka przyporządkowuje



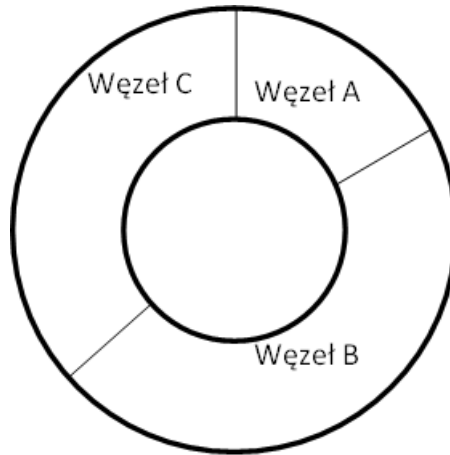
Rysunek 3: Schemat blokowy nowej metody sprawdzania spójności

ciągowi bajtów wartość z pewnego skończonego zakresu w ten sposób że histogram jej zbioru wartości jest „płaski”¹². Ponieważ zbiór wartości klucza jest zbiorem nieskończonym, stosujemy funkcję, która przyporządkuje mu wartość ze zbioru skończonego. Funkcja ta z konieczności musi być funkcją nieodwracalną.

Idea tabeli mieszającej polega na rozdzieleniu między węzły zakresów wartości tej funkcji, dzięki czemu na podstawie klucza ustalić można który węzeł jest zań odpowiedzialny. W praktycznych implementacjach rozproszonych tabel mieszających pojedynczy węzeł nie posiada informacji o wszystkich innych komponentach systemu, a jedynie o niektórych. Każde zapytanie o dany klucz może więc skończyć się dwojako – albo potwierdzeniem, że to właśnie odpytywany węzeł zań odpowiada, albo wskazaniem na inny węzeł, który może mieć tę informację. Przykładem może być tutaj rozwiązanie mapujące znane jako Chord [15], w którym jeden węzeł

¹²w praktyce oznacza to że dla dużej próbki danych wejściowych zaobserwuje się równomierną częstość występowania wartości wynikowych

przechowuje informację tylko o ok. $O(\log N)$ węzłach. Powoduje to że z wysokim prawdopodobieństwem ustalenie węzła następuje w ciągu nie więcej niż $O(\log^2 N)$ wiadomości wymienionych ze składnikami klastra. Istnieją również systemy, w których węzły przechowują informację o wszystkich innych węzłach w klastrze (np. Cassandra [7]).



Rysunek 4: Schemat ideowy przykładowego podziału przestrzeni wartości funkcji mieszającej na trzy węzły

W przypadku systemów w których na jednym węźle dostępna jest pełna informacja o innych węzłach lokalizacja danych zostanie znaleziona najwyżej w 2 zapytaniach. W takim wypadku węzły odpowiedzialne są za przedziały zbioru wartości funkcji mieszającej i tworzą logiczną topologię pierścienia, działającej zgodnie z zasadami arytmetyki modularnej. W tym momencie do stworzenia pełnego obrazu wystarczy, żeby węzłowi przyporządkować wartość początkową przedziału za który jest on odpowiedzialny. Na rysunku 4 został pokazany schemat topologii pierścienia o trzech węzłach. Dodatkową elastycznością takiego rozwiązania jest to, że mocniejszym sprzętowo węzłom można delegować większe zakresy, dzięki czemu możliwe jest zastosowanie takiego rozwiązania w systemach heterogenicznych. Systemem heterogenicznym nazwiemy klaster, w którym węzły nie są identyczne, w szczególności dysponują sprzętem o różnej mocy przerobowej oraz czasami innymi systemami operacyjnymi.

3. Charakterystyka systemu

W rozdziale opisano system kontrolno-pomiarowy, dla którego buduje się rozwiązanie bazodanowe, oraz powody, dla których zdecydowano się na stworzenie nowego rozwiązania. Podano, jak wygląda przepływ danych wewnątrz systemu kontrolno-pomiarowego, oraz analizę, mającą na celu ustalenie ilości danych, z którymi system będzie miał do czynienia. Przedstawiono dalej ogólny charakter tworzonego rozwiązania wraz z jego założeniami, oraz przegląd komponentów systemu. Na końcu przedstawiono właściwości *ciągu*, jako podstawowego elementu projektowanej bazy danych.

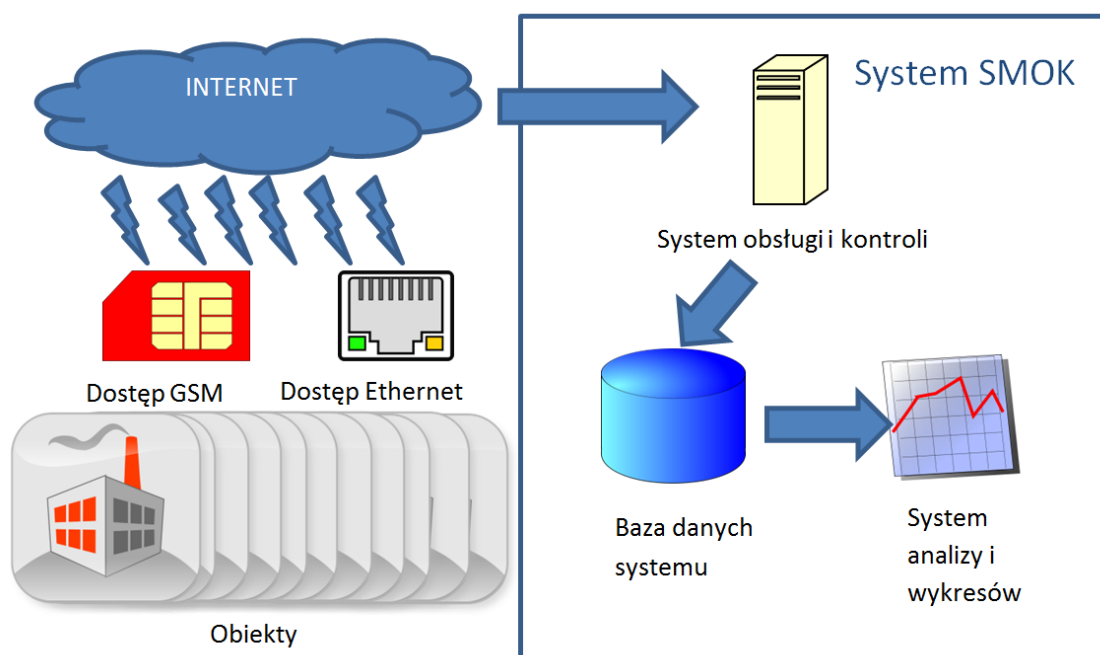
3.1. System kontrolno-pomiarowy

Tworzone rozwiązanie znajdzie zastosowanie w rozproszonych systemie kontrolno-pomiarowym, jakim jest funkcjonujący na rynku od 6 lat **smok-serwis.pl**. SMOK [2] jest tutaj akronimem od „system monitoringu odległych kotłowni“, co jednocześnie jest jego podstawowym zadaniem. System ten zbiera dane z kilkuset instalacji grzewczych, zarówno klasycznych kotłowni gazowych jak i węzłów ciepłych, czy nawet instalacji solarnych, oraz przetwarza te dane, generując powiadomienia o awariach i raporty dotyczące możliwości optymalizacji. Ze względu na ilość zbieranych danych, oraz konieczności zapewnienia nieprzerwanej pracy, nawet w warunkach awarii części infrastruktury serwerowej projektowany system będzie spełniał swoje zadania.

Istotny jest tu warunek poprawnej pracy systemu nawet w przypadku awarii części systemu serwerowego. System w trybie ciągłym musi nadzorować monitorowane obiekty, oraz być w stanie wysłać powiadomienia w wypadku awarii jednego z nich do odpowiednich służb serwisowych. Aby osiągnąć ten cel wymagana jest sprawność podsystemu bazodanowego, co stanowi motywację dla takiego wymagania.

Fundamentalnie SMOK składa się z wielu oddzielnych lub luźno związanych (np. ze sobą obiektów grzewczych (np. węzłów grzewczych na tej

samej linii MPEC¹³), raportujących stan swoich czujników i parametrów do systemu za pomocą modułu dostępu. Medium, za pomocą którego to czyni może być sieć GSM lub lokalne łącze internetowe (Ethernet, WiFi). Moduł dostępu przesyła dane w czasie rzeczywistym do serwerowego systemu obsługi i kontroli, który po ewentualnej obróbce umieszcza dane w bazie systemu kontrolno-pomiarowego. Baza ta następnie służy do prowadzenia analizy i rysowania wykresów. Ideowy schemat architektury systemu SMOK przedstawiono na rysunku 5.



Rysunek 5: Schematyczna architektura systemu SMOK

Mimo że na rynku dostępnych jest szereg rozwiązań bazodanowych, zarówno typu „historian” jak i rodzaju „Big Data”, zdecydowano się na stworzenie nowego rozwiązania. Przemawia za tym szereg powodów:

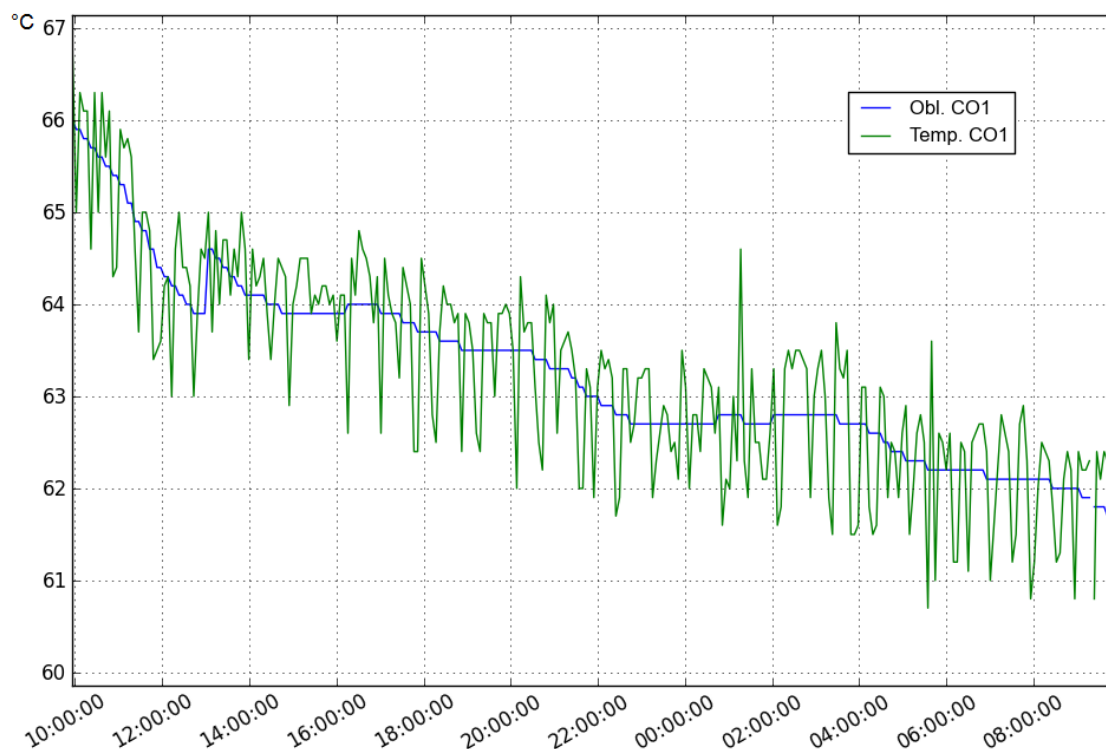
1. Obecne rozwiązania typu Big Data cechuje wysoki poziom trudności przy wdrożeniu. Wymagają one instalacji dużej ilości oprogramowania ze względu na rozległe wymagania elementów.

¹³Miejskie Przedsiębiorstwo Energetyki Ciepłej — funkcjonująca w wielu miastach nazwa lokalnej firmy, będącej zazwyczaj własnością miasta, zajmującej się dostarczaniem energii ciepłej do budynków w postaci ciepłej wody. Ciepła woda ta następnie wykorzystywana jest do ogrzewania budynku za pomocą instalacji ciepłego ogrzewania.

2. Bazy typu „historian“, przystosowane do zbierania danych przemysłowych w sposób ideowo zgodny z zasadami branży, są rozwiązaniami płatnymi. Nowe rozwiązanie planuje się jako otwarte oprogramowanie.
3. Nowe rozwiązanie stosuje nowatorską metodę sprawdzania spójności ciągu, której działanie i przydatność warto zweryfikować.
4. Implementacja wydanej funkcji automatycznego kasowania danych starszych niż zadana wartość jako elementu polityki retencji danych.
5. Elementarnie prosty sposób przechowywania danych pozwala uprościć procedurę odzyskiwania danych w przypadku masowej awarii węzłów.
6. Silna gwarancja przetrwania danych — w momencie gdy baza potwierdzi zapis, dane na pewno nie ulegną utracie w przypadku awarii.
7. Stosowalność i dobry charakter pracy w zaprezentowanym systemie SMOK.
8. Możliwość sprawowania kontroli nad kodem źródłowym i prawami majątkowymi rozwiązania ze względu na brak obciążeń licencyjnych.
9. Zbyt mała ilość danych by angażować typowe rozwiązania Big Data.
10. Zależność innych rozwiązań od dużej ilości innych bibliotek i rozwiązań programowych. Obecność dodatkowych usług i programów może zwiększać podatność na włamania systemu informatycznego.

Podstawowym problemem rozwiązywanym w tej pracy było stworzenie programu — bazy danych. Baza ta ma służyć do przechowywania szeregów pomiarowych. Szereg pomiarowy stanowi zbiór odczytów opatrzonych stemplem czasowym wykonania takiego pomiaru. Jego najprostszą wizualizacją jest wykres obrazujący zmiany parametru w czasie. Szeregi pomiarowe są istotnym rodzajem danych dla systemów kontrolno-pomiarowych.

Baza ta będzie umożliwiała dopisanie do danego ciągu¹⁴ wykonanego pomiaru wraz z jego stemplem czasowym. Będzie możliwe również pobranie historii danego punktu pomiarowego - zarejestrowanych dlań pomiarów wraz z czasami ich zdjęcia. Charakter takich danych najłatwiej przedstawić na wykresie. Przykładowy wykres zaprezentowano na rysunku 6.



Rysunek 6: Przykładowy wykres danych, które mogą być zbierane przez stworzoną bazę. Na wykresie przedstawiono dwa szeregi pokazujące przebieg temperatury w czasie.

Rozwiązanie zamierzono jako rozproszoną bazę danych. Podstawową racją rozproszenia jej jest odporność na awarie pojedynczych węzłów wchodzących w skład klastra bazy oraz skalowalność — to znaczy możliwość obsługi większej ilości danych i połączeń równocześnie niż to możliwe na pojedynczym węźle. W przypadku naprawienia awarii danego węzła i ponownego włączenia go do klastra, węzeł ten powinien zsynchronizować swój stan z resztą, to jest pobrać brakujące dane, które „ominęły” go w trakcie gdy nie był w klastrze.

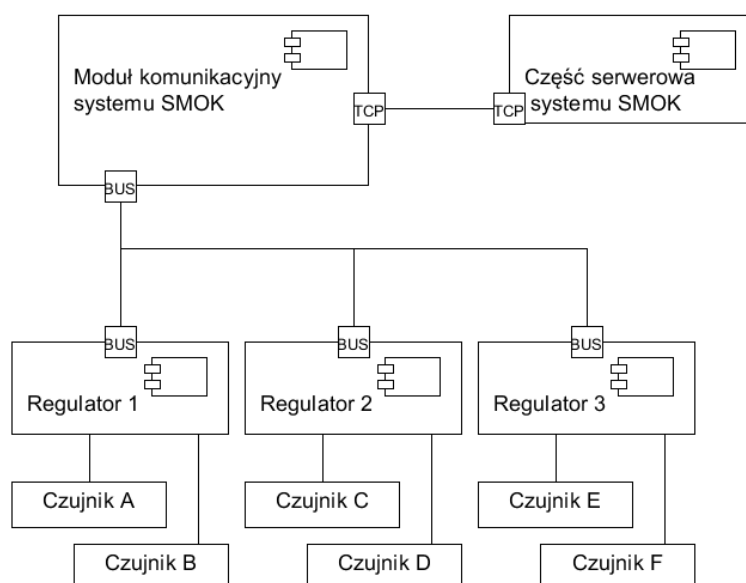
Równocześnie, ze względu na wydajność i tolerancję na awarie, zdecy-

¹⁴pojęcie stosowane w pracy oznaczające szereg pomiarowy

dowano, że w systemie nie będzie węzła typu *master*, który w jakiś sposób mógłby koordynować działania podległych mu węzłów. Ma to istotny wpływ zarówno na projekt bazy, jak i na jej właściwości.

3.2. Przepływ danych

Analizując potrzeby bazodanowe systemu SMOK należy spojrzeć na to, w jaki sposób dane będą przepływać przez system. Należy dokonać takiej analizy zarówno w aspekcie pozyskiwania, jak i odczytywania tych danych. Źródłem informacji są obiekty monitorowane przez system, które definiują listę mierzonych czujników (np. temperatury, ciśnienia). Mierzą one interesujące operatora wartości, takie jak temperatura kotła, temperatura zewnętrzna czy temperatura wody w sieci. Konsumentami informacji natomiast są komponenty optymalizujące pracę kotłowni, generujące raporty, detektory błędów czy interfejs użytkownika Web.

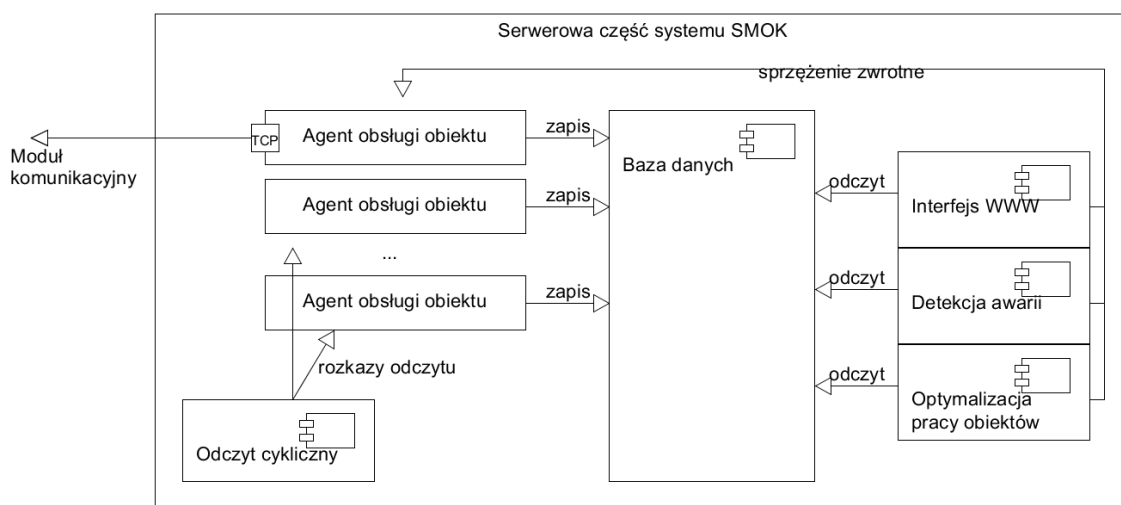


Rysunek 7: Schemat komunikacji w miejscu instalacji modułu

Miejszem najbliższym sensorom mierzącym parametry jest obiekt na którym zainstalowano system. Do regulatorów, zawiadujących pracą urządzeń wykonawczych, podłączone są czujniki. Charakter tego połączenia, z punktu widzenia serwerowej części systemu, jest nieistotny, gdyż system nie prowadzi komunikacji bezpośrednio z nimi. Moduł ten, stanowiący

most między serwerową częścią systemu SMOK a automatyką w miejscu instalacji, koordynuje proces odpytywania konkretnych regulatorów. Dzięki temu moduł komunikacyjny z punktu widzenia serwerowej części systemu stanowi warstwę abstrakcji dla regulatorów i czujników. Ideowy schemat instalacji systemu na obiekcie grzewczym przedstawiono na rysunku 7. Regulatory, zamontowane na obiekcie, są w stanie pracować bez udziału modułu komunikacyjnego. Aby uzyskać odczyt z danego czujnika moduł musi wysłować zapytanie do regulatora. Zapytania takie realizuje moduł na polecenie serwera, z którym łączy się za pomocą protokołu TCP.

Moduł komunikacyjny pracuje w stosunku do serwera w konfiguracji master-slave, realizując rozkazy w reżimie synchronicznym¹⁵. Kolejnym etapem jest rozpatrzenie przepływu informacji wewnątrz systemu serwerowego i jej relacji do systemu bazodanowego. Schemat ideowy przepływu zaprezentowano na rysunku 8.



Rysunek 8: Schemat przepływu danych wewnątrz części serwerowej

Moduł komunikacyjny systemu SMOK łączy się za pomocą połączenia TCP z odpowiednim dla obiektu tzw. agentem obsługi obiektu. Jest to proces działający na systemie serwerowym SMOK. Dla każdego zdefiniowanego w systemie obiektu działa odpowiedni agent. Ze względu na ilość agentów, istotne jest zarządzanie obciążeniem systemu. Agenty bowiem działają równolegle i w izolacji od siebie nawzajem, tak więc nie

¹⁵tj. pytanie-odpowiedź

synchronizują między sobą swoich działań. Nad tą kwestią czuwają nad tym odpowiednie podsystemy które w dynamiczny sposób równoważą obciążenie między węzłami systemu serwerowego, co pozwala systemowi SMOK skalować się liniowo.

Agent obsługi obiektu spełnia dwojaką funkcję. Pierwszą z nich jest odbieranie poleceń odczytu parametrów i modyfikacji parametrów od różnych komponentów systemu. Mogą być to zarówno elementy korzystające z danych, wysyłając takie rozkazy w ramach sprzężenia zwrotnego. Innym elementem, którego jedynym zadaniem jest wysyłanie rozkazów odczytu, jest komponent „Odczyt cykliczny“, który co minutę wymusza odczyt wartości. Robi to, by zapewnić że odczyty realizowane będą w wymaganych odstępach, tj. co minutę. Agent, oprócz poleceń odczytu, otrzymuje też rozkazy zmiany parametrów regulacyjnych na urządzeniu. Dzieje się to zwykle po interwencji użytkownika, choć może być też wykonywane automatycznie poprzez zmiany zaplanowane czy automatyczną optymalizację pracy obiektu.

Drugą funkcją agenta jest wysyłanie poleceń do modułu komunikacyjnego za pomocą połączenia TCP w odpowiednim formacie, oraz koordynowanie ich wykonywania. Odczyty i zapisy wartości agent realizuje szeregowo, w takim formacie bowiem przyjmuje je moduł komunikacyjny. Wykonanie serii odczytów zajmuje pewien czas. Ponieważ system odczytuje parametry co minutę, czas trwania serii odczytów musi być mniejszy od minuty. W praktyce jest on zupełnie wystarczający, nawet na odczytanie kompletnego zestawu pomiarów z bardzo dużych instalacji¹⁶.

Agent po otrzymaniu wartości parametru od modułu komunikacyjnego umieszcza go w bazie danych. Odpowiada on za wszystkie ciągi wartości pomiarowych danego obiektu i jest jedynym elementem w systemie który będzie do tych ciągów dopisywał nowe dane. Z danych tych później korzystają inne elementy systemu. Największymi konsumentami tych danych są trzy podzespoły uwidocznione na rysunku 8. Są to:

¹⁶rzędu czterech kotłów gazowych, dziesięciu obiegów centralnego ogrzewania i trzech zasobników ciepłej wody

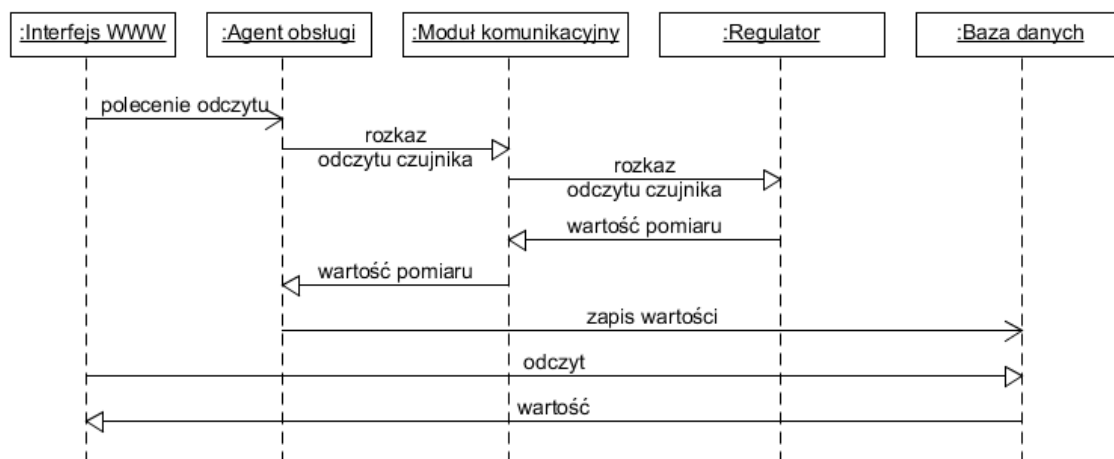


Rysunek 9: Zrzut ekranu interfejsu WWW z produkcyjnej wersji systemu SMOK

- interfejs WWW — zapewnia dostęp do informacji odnośnie stanu obiektu przez WWW. Pozwala użytkownikowi przeglądać aktualny stan, uzyskiwać dostęp do wykresów obrazujących jego pracę oraz pozwala zmieniać parametry regulacyjne. Przykładowy zrzut ekranu z produkcyjnej wersji systemu przedstawiono na rys. 9,
- detekcja awarii — jest to podzespół który niezależnie od kontroli obiektów przez użytkowników wykrywa czy parametry pracy nie świadczą o problemach regulacyjnych bądź awarii elementów wykonawczych. Po stwierdzeniu awarii informuje on użytkownika o problemie,
- optymalizacja pracy obiektów — podzespół ten analizuje pracę obiektu i jest w stanie interweniować w przypadku wykrycia sytuacji w której automatyka reguluje układem w sposób nieoptymalny ze względu na źle dobrane parametry regulacji (np. nastawy regulatorów PID).

Agent obsługi obiektu dopisuje do bazy, wartości pomiarów zaraz po ich odebraniu, które z drobnymi odstępami czasowymi odbiera szeregowo. Interfejs WWW odczytuje najczęściej ostatnią dopisaną wartość do danego ciągu, czasami całe zakresy czasowe danych w przypadku konieczności stworzenia wykresu. Pozostałe podzespoły odczytują całe zakresy czasowe.

Ze względu na możliwość prowadzenia różnych przekształceń przez agenta, polecenia odczytu kierowane do agenta są tylko asynchroniczne. Przechodzą one dość długą drogę, co ilustruje diagram UML sekwencji na rysunku 10. Agent przed wykonaniem otrzymanego polecenia może wykonywać również inne operacje, co decyduje o asynchroniczności części operacji.



Rysunek 10: Schemat sekwencji odczytu pojedynczego parametru

Powyższe przedstawienie przepływu danych wewnątrz systemu kontrolno-pomiarowego, z którym ma być sprzężona projektowana baza danych, uwidacznia charakter operacji przeprowadzanych na tej bazie. Dodatkowo pokazuje dlaczego stabilne i pewne działanie bazy jest konieczne dla pracy systemu.

3.3. Analiza ilości danych

Aby system mógł wywiązywać się z tych zadań, musi przechowywać i przetwarzać szeregi czasowe. Są to parametry z podłączonych kotłowni, takie jak stan urządzeń wykonawczych, temperatury, ciśnienia czy natężenia prądów. W przypadku komunikacji GSM przechowywana jest także informacja o jakości połączenia. Prezentowana w tej pracy baza danych będzie stanowić podsystem odpowiedzialny właśnie za to. Aby przedstawić konkretny plan, niezbędna jest najpierw analiza wolumenu danych, które system przetwarza obecnie, oraz prognozy wzrostu.

Wg stanu na 18 grudnia 2014, SMOK monitoruje 512 obiektów. Każdy

z nich ma średnio 14 punktów pomiarowych, z których dane muszą być archiwizowane bezterminowo, oraz około 98 punktów, które mogą być przechowywane tylko przez krótki okres¹⁷. Istnieją również dodatkowe informacje na temat obiektów, jednak nie są one interesujące z punktu widzenia analizy ilości zebranych danych.

$$512 * 14 = 7168$$

$$512 * 98 = 50176$$

Sumarycznie daje to 7168 punktów przechowujących dane bezterminowo i 50180 punktów o krótkim przechowywaniu danych. Każdy z nich wymaga maksymalnie 4 bajtów na opisanie wartości, opowiadającej zmiennej typu *float*. Po doliczeniu stempla czasowego, jeden rekord będzie miał 12 bajtów. Pomiar przechowywany bezterminowo generowany są co jedną minutę. W przypadku pomiarów o przechowywaniu terminowym będzie to około 15 sekund, ale przy przechowywaniu jest to nieistotne na dłuższą metę, gdyż zgodnie z założeniami ma być przechowywane maksymalnie 15 minut.

Zapis co minutę jest z punktu widzenia analizowanych zjawisk wystarczający. SMOK zajmuje się zagadnieniami ogrzewania. Procesy cieplne są procesami wolnozmiennymi. Zjawiska okresowe, mogące być interesujące z punktu widzenia analizy, mają okres właściwie zawsze większy od 2 minut, co spełnia warunek Nyquista. Patologicznymi zjawiskami okresowymi są np. oscylacje temperaturowe, świadczące o błędnej regulacji układu. W procesach inercyjnych, jakimi są procesy temperaturowe, są to procesy wolnozmiennne. Zbieranie wartości co minutę w zupełności wystarcza, aby zarejestrować i zidentyfikować takie przypadki. Ponadto doświadczenie z ponad pięcioletniej eksploatacji systemu dodatkowo upewnia w słuszności powziętego założenia.

Tak więc ciągi generowane przez punkty o przechowywaniu terminowym

¹⁷Dużą część z nich stanowią parametry regulacyjne, które nie w każdym cyklu są odczytywane. Inne odczyty nie są tak ważne by archiwizować je bezterminowo, jednak są przydatne operatorowi nadzorującemu pracę obiektu po zalogowaniu do systemu.

zajmą maksymalnie 35 MB¹⁸. Ciągi o przechowywaniu bezterminowym generują ok. 84 kB¹⁹ danych na minutę, co tłumaczy się na 120 MB dziennie, czyli 42,8 GB rocznie²⁰. Tak więc — ze względu na takie porównanie — w dalszej analizie można zaniedbać rozpatrywanie ciągów o przechowywaniu terminowym. Wg zgłoszonego przez operatora stanu na 18 grudnia 2014, system zgromadził 48 GB danych dla 512 urządzeń.

System ulega ciągłej rozbudowie. Statystyki udostępnione przez operatora systemu wskazują, że jedno urządzenie dostępowe, obsługujące nowy obiekt grzewczy, montowane jest średnio co 3 dni. Wykonano dodatkowo prognozy dla montowania urządzenia co 2²¹ oraz co 5 dni. Są to wartości odpowiednio uśrednione, użyteczne dla zastosowań modelowania i prognozy.

Jako okres obliczania można więc przyjąć x dni. Tak więc, ilość danych zgromadzonych w ciągu takiego okresu wynosić będzie

$$x(14 \times 12B \times 4min^{-1} \times 60 \frac{min}{h} \times 24h) = x \times 80640B$$

Mając początkową wartość 48 GB w okresie „startowym” o_0 ilość danych dla następnego okresu obliczyć można:

$$o_{a+1} = o_a + x \times p \times 80640B$$

Gdzie p to liczba urządzeń. Oczywiście, będzie się ona zwiększać co okres rozliczeniowy. Mając początkową liczbę urządzeń równą 512 oznaczoną p_0 można zapisać:

$$o_{a+1} = o_a + (p_0 + a) \times x \times 80640B$$

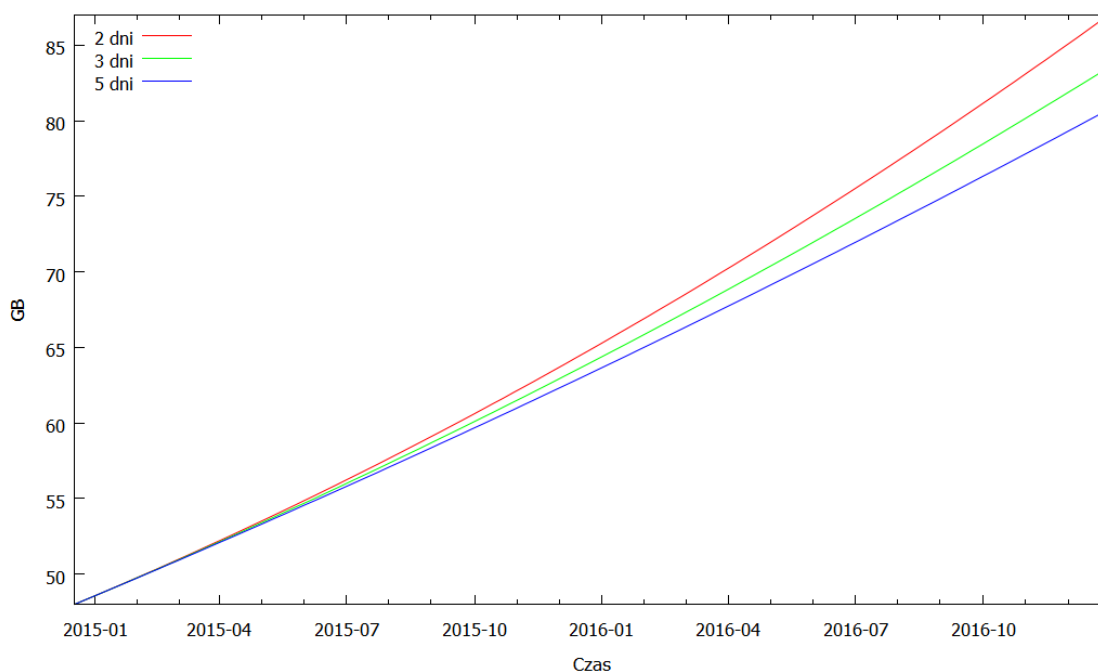
Wykonaną za pomocą tego modelu prognozę na rok 2015 i 2016 ujęto w wykresie na rysunku 11. Jest to oczywiście tylko prognoza, nieuwzględniająca zmienności warunków rynkowych. Na koniec roku 2016 przewiduje się rozmiar danych najwyżej 87 GB. Przy założeniu podwójnej replikacji

¹⁸ $50180 \times 12b \times 4min^{-1} \times 15min$

¹⁹ $7168 \times 12b$

²⁰jest to wartość mniej więcej zgodna z rzeczywistością, danych tych jest nieco mniej ze względu na awarie poszczególnych urządzeń

²¹Wg operatora systemu występowały przypadki montażu nawet 20 urządzeń dziennie.



Rysunek 11: Prognoza ilości danych zgromadzonych w systemie na lata 2015, 2016 wykonana za pomocą modelu dla trzech różnych odstępów między dodaniem nowego urządzenia do systemu.

danych będzie to zajmować 174 GB danych. Stworzona baza danych przeszła integrację na testowej platformie systemu, zaś jej szersze wdrożenie planowane jest na wiosnę 2015.

Aby dokładniej określić wymagania systemu, należy ustalić maksymalne planowane obciążenia systemu. W tym przypadku, biorąc pod uwagę odpowiednio duży zapas, zdecydowano się na ustalenie tej wartości jako model 2-dniowy²² dla roku 2025. Jest to 2345 urządzeń oraz 442 GB zgromadzonych danych, co dla podwójnej replikacji daje 884 GB danych. Planując cztery węzły, na każdym wymagane jest przynajmniej 221 GB na ten cel. Daje się to zrealizować na typowych dyskach 500 GB czy nawet 250 GB. Nie muszą być to systemy RAID, ze względu na redundancję systemu.

²²to jest, przy założeniu dodawania jednego obiektu co w dni

3.4. Założenia i zarys działania

Pierwszym założeniem, które należy poczynić przy projektowaniu jakiegokolwiek bazy danych jest charakter przechowywanych danych. Projektując bazę systemu kontrolno-pomiarowego wiadomo że będą to dane pomiarowe, czyli zbiór uporządkowany wartości opatrzonej informacją o jakości próbki i czasie jej pobrania. W bazie występować może wiele takich ciągów, z których każdemu, w celu identyfikacji, nadać należy nazwę.

Oczywiste również jest że wartość stempla czasowego będzie stale rosnąć. Istnieje zależność polegająca na tym, że w ramach danego przyporządkowania – dalej nazywanego ciągiem – wartość stempla będzie rosnąca. Niemożliwe będzie dodanie do danego ciągu próbki starszej niż najmłodsza w nim obecna. W znaczący sposób ułatwi to później przeszukiwanie i zarządzanie danymi.

Dodatkowo każda próbka, którą wraz ze stemplem czasowym nazwać można rekordem, będzie miała taką samą długość. Systemy pomiarowe z reguły generują dane o równej długości, najczęściej będące właśnie krotką stempla czasowego, wartości zmierzonej i informacji o jakości próbki. Nadanie im równych długości w ramach konkretnego ciągu pozwoli na istotne uproszczenie i przyspieszenie działania systemu. Przykład takiego ciągu zamieszczono na rysunku 12.

	Rekord 1	Rekord 2	Rekord 3	Rekord 4	Rekord 5
Stempel czasowy	23:11	23:13	23:15	23:17	23:19
Wartość	12°C	11.5°C	11.5°C	11°C	10.5°C

Rysunek 12: Przykład ciągu o 5 rekordach z rosnącym stemplem czasowym. Ze względu na czytelność datę pominięto.

Reasumując, możliwymi operacjami do wykonania na ciągu muszą być przynajmniej odczyt, dopisanie nowego rekordu i skasowanie całego ciągu. Jeśli wymagać tylko tych operacji, to takie ograniczenie pozwoli w dalszym rozrachunku na uproszczenie programu i jest zupełnie wystarczające dla celów bazy danych systemu kontrolno-pomiarowego.

Baza musi także zapewnić, że awaria jednego węzła nie wpłynie katastroficznie na działanie całego klastra. W rozdziale 2 wymieniono możliwe podejścia do zarządzania takim klastrem: zdecydowano się na podejście bez węzła typu master. Wszystkie węzły są równorzędne – co jest jednym ze sposobów na zapewnienie braku pojedynczego punktu awarii w systemie. Jakość pracy bazy, zarówno w warunkach normalnych jak i przy częściowej awarii nie może powodować odrzucania zapisów czy uniemożliwienia odczytów zgromadzonych danych. Częściowa awaria klastra może co najwyżej spowodować degradację jakości usługi. Degradacja ta przejawiać może się zwiększeniem czasu wykonywania operacji, ale nigdy utratą danych czy niemożnością odczytu bądź zapisu.

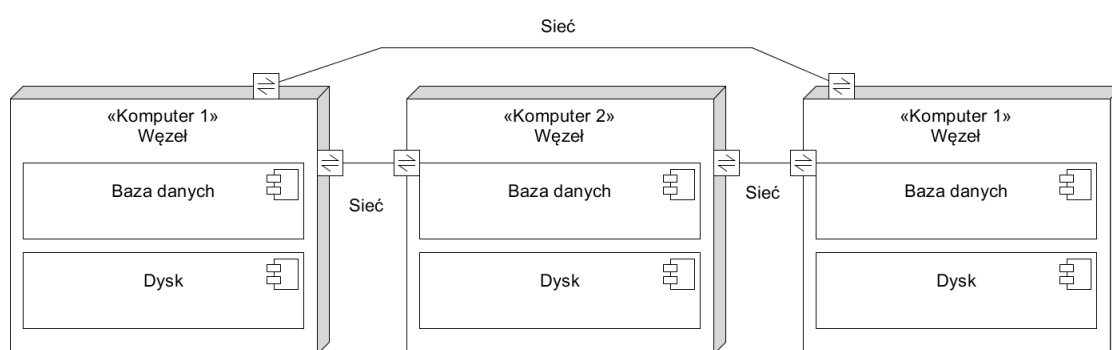
Założeniem szczegółowym do wcześniej wymienionego jest to, aby żądania zapisu nie były przetwarzane zbyt długo - zwłaszcza nie powinna na czas zapisu wpływać awaria innych węzłów. Możliwe jest, że klient zdecyduje się na dość duży wolumen zapisywanych danych, przez co opóźnienia są niedopuszczalne. Wszelkie procesy naprawcze, jeśli są wymagane, powinny przebiegać w tle, zaś węzeł zobligowany jest do przyjmowania rozkazów zapisu.

W rozdziale 2 wspomniano także o wybranej metodzie utrzymywania spójności replikowanych ciągów na węzłach. Dodatkowym założeniem będzie więc, że element dopisujący rekordy do konkretnego ciągu będzie musiał pamiętać zarówno definicję ciągu, jak i stempel czasowy pod którym dodał ostatnią informację.

Dodatkowym założeniem jest przenośność systemu. Baza powinna być uruchamialna na dowolnym systemie obsługiwanym przez technologię, która umożliwi taką przenośność. W szczególności powinna umożliwiać konfigurację klastra w systemach heterogenicznych, tj. dysponujących różnymi systemami operacyjnymi i parametrami sprzętowymi. W celu zapewnienia zarówno przenośności jak i odpowiedniej prędkości działania systemu zdecydowano się na wykonanie go przy użyciu języka Java w szeroko wspieranym wydaniu Standard Edition.

Ze względu na wspomnianą wcześniej możliwość pracy w systemach

heterogenicznych zakłada się że węzły te nie będą miały bezpośredniego dostępu do współdzielonej przestrzeni dyskowej. Każdy węzeł może uzyskiwać dostęp jedynie do swojej przestrzeni dyskowej. Jedyną metodą komunikacji międzywęzłowej są połączenia sieciowe realizowane przez bazę danych. Podjęcie takiego założenia przyczyni się także do uproszczenia zarządzania infrastrukturą klastra, zawężając wpływ czynników zewnętrznych na działanie węzłów. Innymi słowy, programy-węzły znajdować będą się na różnych fizycznych maszynach, jak przedstawiono na rysunku 13.



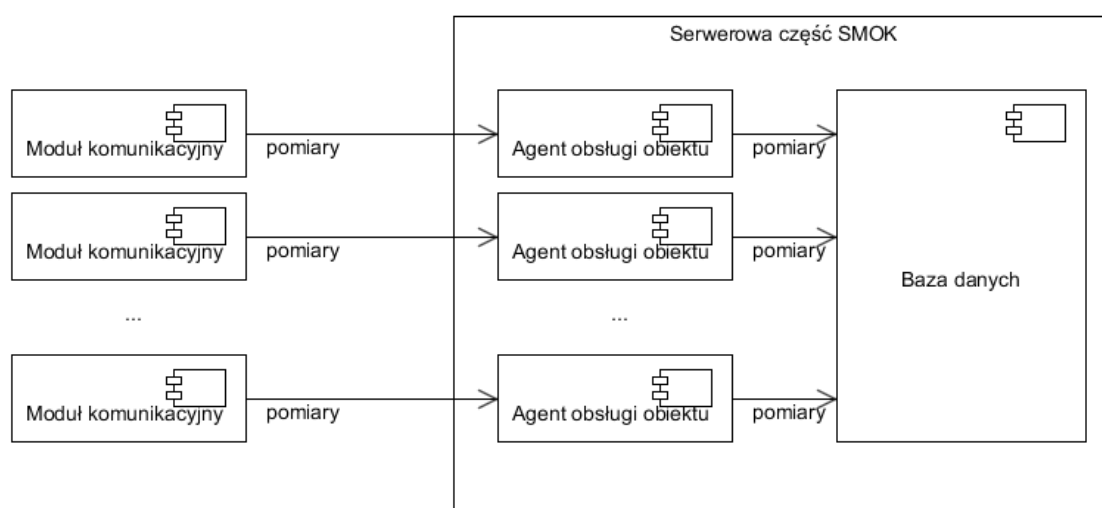
Rysunek 13: Diagram struktury przykładowej instalacji z trzema węzłami

W rozważaniu zagadnienia, jakim są bazy danych, niezwykle ważnym aspektem jest skalowalność. Wymaganiem jest, aby baza była skalowalna zarówno poziomo jak i pionowo, liniowo w funkcji dostępnych zasobów sprzętowych. Spełnienie tego założenia wymaga konkretnych decyzji natury projektowej, które uwzględniono w strukturze bazy.

Istotnymi założeniami są również te natury infrastrukturalnej. Jako rozwiązanie programowe, do wdrożenia klastra niezbędne są konkretne zasoby sprzętowe, w stosunku do których istnieją również pewne wymagania. Przede wszystkim system (to jest interfejs oraz wszystkie węzły) musi działać w całości w zaufanej sieci lokalnej. Wszystkie hosty w tej sieci mają dostęp do odczytu i zapisu do innych węzłów i mogą łączyć się swobodnie ze sobą (muszą mieć realizacji logicznej topologii pełnej siatki). Musi być to sieć zaufana, gdyż węzły nie będą przeprowadzać uwierzytelniania, a napastnik z dostępem do sieci lokalnej byłby w stanie skutecznie sparaliżować pracę klastra. Elementy zapisujące dane w takiej

bazie również musiałyby należeć do zaufanej sieci lokalnej. W przypadku geograficznego rozproszenia węzłów sieci można zastosować technologie wirtualizujące środowisko lokalne, takie jak VPN. Spełniają one bowiem warunek zaufanej sieci.

Ze względów organizacyjnych należy założyć także, że do danego ciągu zapisywać będzie tylko jeden element zespołu. Biorąc pod uwagę wdrożenie bazy jako systemu kontrolno-pomiarowego, jest to założenie logiczne i naturalne. Jeśli ciąg ewidencjonuje konkretny punkt pomiarowy w systemie, to jest to punkt unikatowy i z pewnością będzie istniał jeden element zapisujący ten ciąg. Jak wspomniano wcześniej, dysponuje on wiedzą na temat ciągu, który zapisuje. Wynika to z konstrukcji systemu, w której wiele modułów komunikacyjnych zainstalowanych na monitorowanych obiektach łączy się z niezależnie i równolegle pracującymi agentami po stronie serwera. Agenty te następnie zapisują zebrane dane do bazy danych. Schematycznie przedstawiono tą sytuację na rysunku 14. Ponieważ pojedynczy agent odpowiada za wszystkie ciągi danego obiektu, założenie to jest spełnione.

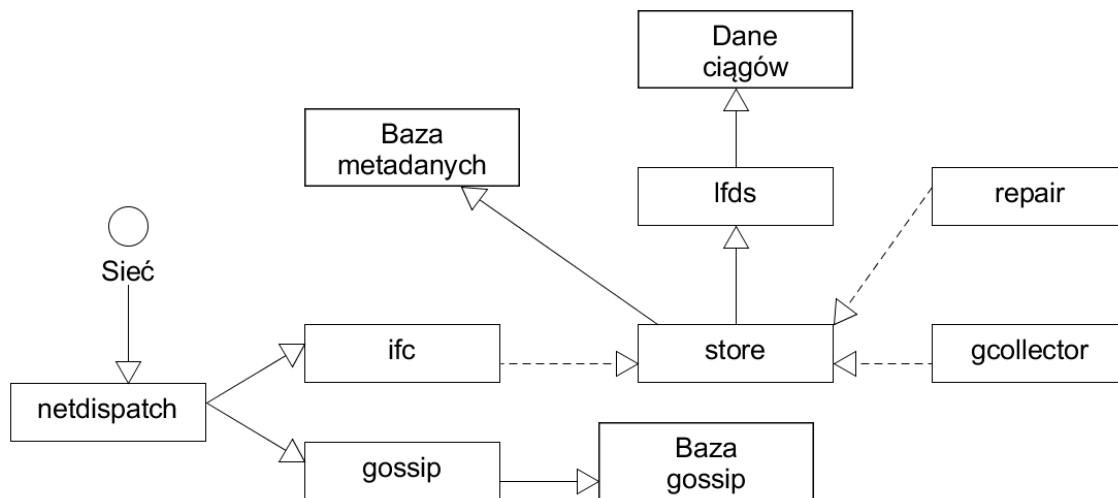


Rysunek 14: Diagram ogólnej idei przepływu danych

Ze względu na uproszczenie projektu zdecydowano się, aby założyć, że dany ciąg w całości zmieści się na jednym węźle. Nie jest to wymóg w żaden sposób przesadzony – wstępne szacunki zakładają że do przechowania roku pomiarów wartości zmiennoprzecinkowej zdejmowanych co 5 sekund

nie powinno być potrzebne (zaniedbując dodatkowe narzuty) więcej niż 72 MB²³.

Baza jako system modułowy i mający wiele aspektów działania składa się z współpracujących ze sobą elementów. Każdy z nich zapewnia konkretną funkcjonalność, jednak dopiero razem stanowią zgodny z założeniami program. Ogólny diagram modułów i ich powiązań przedstawiono na rysunku 15. Projektowane rozwiązanie składa się z następujących modułów:



Rysunek 15: Prezentacja modułów i ich wzajemnych powiązań. Pogrubione ramki oznaczają repozytoria danych, strzałki linią przerywaną oznaczają relatywnie „luźniejsze” sprzężenie.

- *netdispatch* odpowiada za przyjmowanie połączeń sieciowych za pomocą gniazdek TCP, a następnie przekazywanie ich do dwóch modułów które korzystają z łączności sieciowej,
- *gossip* ma za zadanie koordynację z innymi węzłami i utrzymuje w danym węźle reprezentację innych węzłów w klastrze — ich adresów, stanów awarii i konfiguracji. *Gossip* wymienia się tymi danymi z innymi węzłami w określony sposób, zapewniając aktualność danych. Ogólny opis jego działania podano w p. 3.5,

²³Założono 8 bajtów na znacznik czasu oraz 4 bajty na wartość.

- *ifc* stanowi interfejs między binarnym protokołem sieciowym a modułami wykonawczymi systemu. Wykonuje on komendy zadane przez klienta przy użyciu modułu *store* i odsyła wyniki. *ifc* odpowiada również za przekierowanie rozkazu, jeśli okaże się że węzeł z jakichś przyczyn nie może wykonać otrzymanego rozkazu,
- *store* to moduł koordynujący dostęp do danych zgromadzonych na węźle. Zapewnia on funkcjonalności dotyczące tworzenia oraz kasowania ciągów. Zawiera także interfejsy do dodawania i odczytywania danych pomiarowych,
- *lfds* odpowiada za fizyczną, dyskową reprezentację zgromadzonych danych. Dbą o to, aby w szybki sposób można było odnaleźć żądane wycinki danych oraz o zagadnienia synchronizacji dostępu,
- *gcollector* zapewnia wsparcie dla kasowania ciągów. Ze względu na założony model bazy danych jest on niezbędny aby kasowanie mogło być dokonane w sposób zsynchronizowany na wielu węzłach jednocześnie,
- *repair* dokonuje napraw danych ciągu, jeśli zostanie wykryty brak spójności danych. Działa on w tle, do minimum redukując zakłócenia związane z swoją pracą.

3.5. Kluczowe problemy i ich rozwiązanie

Ponieważ zdecydowano się na zarządzanie rozproszeniem bazy poprzez uczynienie wszystkich węzłów równorzędnymi, pojawia się dodatkowy problem. Mianowicie brak jest centralnego katalogu, który przechowywałby informację o węzłach obecnych w systemie, ich stanie oraz informacji, które pozwolą ustalić, który węzeł odpowiedzialny jest za konkretny ciąg. Informacje takie są kluczowe, zwłaszcza w momencie, gdy system otrzymuje żądanie odczytu bądź zapisu. Musi on wtedy, w szybkim tempie, ustalić, do którego węzła trafi polecenie.

Zdecydowano się tutaj na podobne rozwiązanie jakie zostało zastosowane w bazie danych Cassandra [7] pod nazwą *gossip*. Rozwiązanie to polega na wymienianiu się przez węzły fragmentarycznymi informacjami o stanie swoim i innych węzłów. Przy wystarczającej ilości takich wymian dany węzeł będzie znał stan wszystkich węzłów w danym klastrze. Wymienianie się takimi komunikatami zgodnie z pewnymi regułami zapewnia, że aktualizacje stanu będą propagowane zarówno szybko, jak i do wszystkich innych węzłów.

Każdy węzeł dysponować będzie informacjami o adresach i znanym stanie każdego innego węzła w klastrze. Taka struktura danych nie musi być większa niż ok. 200 bajtów na węzeł, co w przypadku nawet dużych klastrów nie będzie w znaczący sposób wywierać istotnego negatywnego wpływu na zużycie pamięci. Celem zainicjowania takiej tabeli przy ruchu węzła podawać będzie się mu adres innego węzła, który będzie dysponował już jakimiś informacjami nt. klastra (tj. będzie już weń włączony). Taki węzeł startowy następnie roześle informację innym węzłom o nowym elemencie klastra.

Przy jakiegokolwiek modyfikacji stanu klastra, czy wykryciu że jeden z węzłów uległ awarii, taki system równorzędnej wymiany wiadomości o stanie klastra sprawi że w skończonym czasie wszystkie składowe będą miały spójny obraz o stanie całości systemu. Dzięki temu będą w stanie na jego podstawie podejmować decyzje. Warto jednak zadbać o to, aby nie zdarzyła się sytuacja, w której dwie aktualizacje stanu tego samego węzła będą ze sobą konkurować. Każda informacja o danym węźle musi być opatrzona znacznikiem czasowym, nadawanym przez wysyłającego. Wtedy, węzeł odbierając dwie nawet sprzeczne informacje, będzie w stanie zaktualizować swoją bazę tą nowszą, zapewne bardziej zbieżną z rzeczywistością, informacją.

Istotnym problemem, który rozwiązano jest metoda mapowania ciągów oraz ich kopii zapasowych na węzły. Rzecz dotyczy się tego, jak po nazwie, czy też definicji ciągu, określić węzeł, na którym się znajduje. Jak wcześniej ustalono, klaster pracować będzie bez węzła typu master, tak więc


```

/**
 * Hashes the name.
 * @param name Name to hash
 * @param replica_no Number of replica, counted from 0
 * @return Hash value
 */
public static long hash(String name, int replica_no) {
    long basehash = H.h_hash(name);
    if (replica_no == 0)
        return basehash;
    if (replica_no == 1)
        return basehash + Long.MAX_VALUE;
    if (replica_no == 2)
        return basehash + (Long.MAX_VALUE) + (Long.MAX_VALUE / 2);
    if (replica_no == 3)
        return basehash + (Long.MAX_VALUE / 2);
    throw new RuntimeException("No idea how to hash that");
}

```

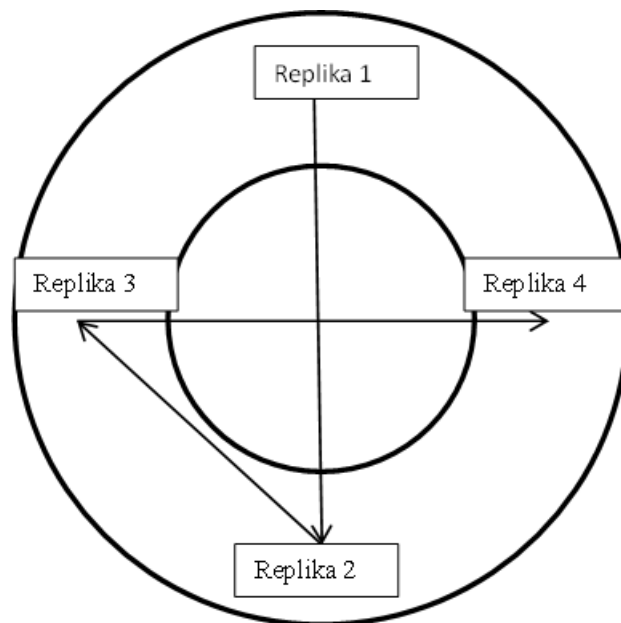
Listing 1: Funkcja mieszająca biorąca pod uwagę także numer repliki danej

nie istnieje centralny katalog ewidencjonujący wszystkie ciągi obecne w systemie. Zdecydowano się rozwiązać to za pomocą systemu mieszania opisanego w rozdziale 2.4.

Oczywiście rozwiązanie to w podstawowym formacie nie będzie współgrać z założeniem redundancji danych. Aby określić dodatkowe węzły odpowiedzialne za dany ciąg, można dodać do wartości funkcji mieszającej pewną stałą wartość. Sprawia ona, że oznaczony zostanie węzeł znajdujący się na „przeciwnej” stronie pierścienia wartości – więc z pewnością inny niż ten, który uległ awarii. Taki sposób mieszania ilustruje funkcja wchodząca w skład bazy, przedstawiona na listingu 1. Graficzna interpretacja takiego sposobu przedstawiona jest na rysunku 16. Z przyczyn praktycznych ograniczono maksymalną ilość replik²⁴ do czterech. Nie zakłada się, że będą definiowane więcej niż 4 repliki danego ciągu. Dodanie większej liczby wymaga po prostu uzupełnienia dodatkowych współczynników w funkcji mieszającej, więc takie ograniczenie nie będzie to stanowiło praktycznego problemu.

Kolejne zagadnienie dotyczy definiowania ciągów. Aby utrzymać spójność systemu, każdy z węzłów musi dysponować identyczną definicją.

²⁴kompletnej kopii wszystkich danych konkretnego ciągu



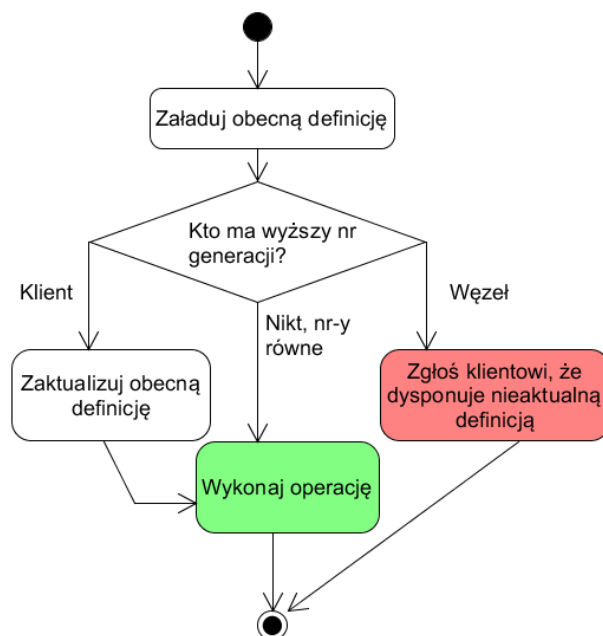
Rysunek 16: Graficzna interpretacja przydziału węzłów odpowiedzialnych za repliki, w zbiorze wartości funkcji mieszającej

Możliwa jest także sytuacja, w której węzeł odpowiedzialny za jedną z replik w chwili tworzenia ciągu, nie będzie dostępny i ominie go komenda stworzenia ciągu. Podobna sytuacja może się zdarzyć w przypadku zmiany definicji ciągu.

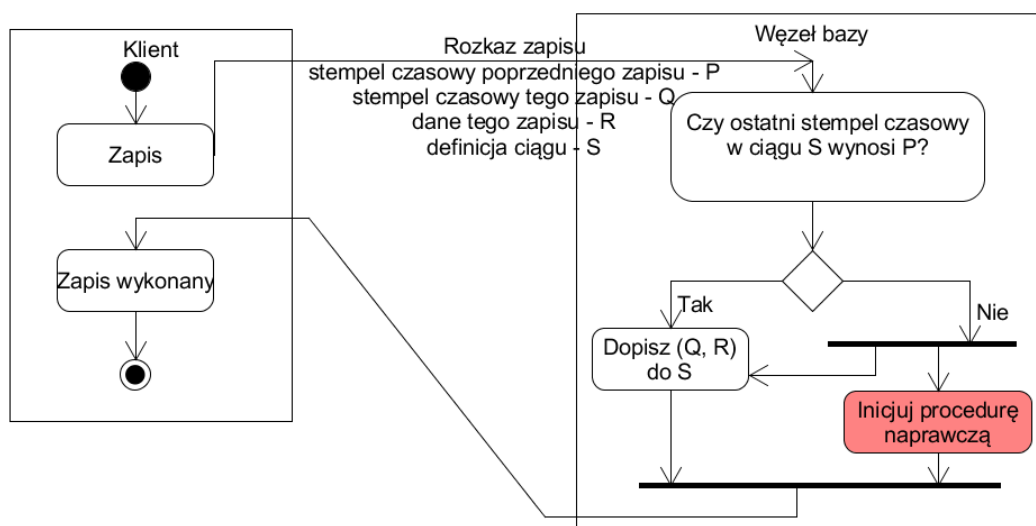
Problem rozwiązano w nietypowy sposób, poprzez obowiązek pamiętania przez klienta definicji ciągu. Choć baza zapewnia możliwość pobrania definicji po nazwie ciągu²⁵, jednak przy zapytaniach odczytu bądź zapisu klient musi podać kompletną definicję ciągu. Oczekuje się, że klient będzie trzymał później tę definicję w pamięci podręcznej. Dzięki temu, jeśli ciąg nie istnieje, będzie mógł być utworzony, a dane pobrane z innych węzłów dysponujących replikami. Dodatkowo każda definicja ma pole *generacji*. Jest to pole liczbowe określające która „wersja” definicji jest podana. Jeśli węzeł dysponuje starszą, zostanie zastąpiona nowszą — podaną przez klienta lub inny węzeł. Algorytm takiego zamieniania przedstawiono na rysunku 17. Dzięki takiemu rozwiązaniu węzły obsługujące repliki danego ciągu są w stanie skutecznie ustalić bieżącą definicję ciągu. Przewiduje się, że nie będzie się ona często zmieniać, a nawet jeśli, to dzięki skutecz-

²⁵określonej jako ciąg znaków

nemu systemowi numerowania wersji (generacji) definicji klaster będzie w stanie utrzymać spójność. Każda nowa redefinicja będzie musiała mieć zwiększony numer generacji.



Rysunek 17: Algorytm porównania i synchronizacji definicji



Rysunek 18: Algorytm detekcji niespójności ciągu

Oczekuje się, że pamiętanie przez klienta w trakcie sesji kompletnej definicji ciągu nie będzie wielką przeszkodą. Zwykle to jeden klient dopisuje dane do jednego ciągu, więc ryzyko kolizji jest zredukowane do minimum, zwłaszcza jeśli projektant systemu przechowywania danych

podejmie decyzję że tylko klient zapisujący jest uprawniony do redefinicji ciągu. Wymaganie to jest spełnione w przypadku systemu SMOK, gdzie takim klientem jest agent obsługi obiektu.

Innym problemem jest kasowanie ciągów. Ze względu na fakt, że jeden z węzłów obsługujących dany ciąg może być w trakcie awarii w czasie wydania takiego polecenia, komenda skasowania ciągu może go ominąć. Gdyby rozwiązanie to wykonać w sposób naiwny, to gdy taki węzeł zostanie naprawiony zreplikuje on innym węzłom „skasowany” ciąg. Oznaczaoby to, że procedura usuwania ciągu będzie bezskuteczna.

Rozwiązano to, dodając do definicji dodatkowe pole. Pole to informuje o tym, kiedy i czy dany ciąg został usunięty. Wzorowano się tutaj na rozwiązaniu o nazwie *tombstone* zastosowanym w bazie Cassandra [7]. Tak więc kasowanie ciągu polega na zmianie jego definicji ustawiając wartość tego pola na obecny czas. Taki ciąg nie będzie replikowany innym węzłom, zaś jego dane (wartości pomiarów) zostaną skasowane, choć metadane pozostaną. Po upływie tzw. czasu bezpiecznego (np. tydzień) podsystem usuwania „starych” metadanych taki wpis usunie. Rodzi to pewne problemy o których szerzej wspomniano w rozdziale 5.4.

Kolejnym problemem była detekcja spójności ciągu. W momencie kiedy pojedynczy ciąg śledzony jest jednocześnie na kilku węzłach — w formie replik — możliwym zjawiskiem jest to, że na pewien czas węzeł posiadający drugą, czy nawet trzecią replikę będzie zmuszony do odebrania tych zapisów jako jedyny. Pozostałe węzły, które w tym czasie były offline, muszą dysponować metodą przywrócenia poprzednich danych gdy wykryją niespójność.

Zdecydowano się na rozwiązanie tego w nietypowej formie, niespotykanej w obecnych systemach bazodanowych. Mianowicie, każde żądanie zapisu, zawierające koniecznie rekord, zawiera również stempel czasowy poprzedniego zapisu. Klient, ze względu na strumieniowy charakter swojego zapisu, dysponuje taką informacją (zgodnie z zaleceniami z rozdziału 5.5). W tym momencie, węzeł otrzymując żądanie zapisu, sprawdza czy rzeczywiście dysponuje rekordem o wartości stempla przesłanej jako po-

przednia. Jeśli ten rekord istnieje, to można stwierdzić, że ciąg którym dysponuje węzeł jest spójny i kontynuować pracę. Jeśli nie zostanie on odnaleziony, należy zainicjować procedury naprawcze. Algorytm ten przedstawiono na rysunku 18.

3.6. Anatomia ciągu

Ciąg, jako podstawowy element logiczny systemu, wymaga kilku słów opisu i wyjaśnienia. Jako element złożony, podzielić można go na dwie części — dane oraz metadane. O formule danych wspomniano już w rozdziale 3.4, tak więc fakt przechowywania ich jako uporządkowanych po czasie par stempel czasowy-wartość jest oczywisty. Kłopotem może być dokładne oznaczenie pól metadanej ciągu. Zdecydowano się na następujące składowe takiego obiektu:

- *seriesName* — nazwa ciągu. Jest to podstawowy identyfikator danego ciągu w obrębie klastra. Dysponując nazwą można wystosować zapytanie o pobranie metadanych danego ciągu, tak więc stanowi ona taki „punkt wejścia” do danego ciągu,
- *replicaCount* — ilość replik danego ciągu. Dysponując taką informacją można od razu ustalić ile węzłów będzie musiało zostać poinformowanych o zapisie nowego rekordu czy zmianie definicji ciągu,
- *generation* — numer generacji danego ciągu. Za każdym razem, kiedy jego definicja się zmienia, ta wartość jest zwiększana o jeden. Dzięki temu, gdy węzły odpowiedzialne za dany ciąg dysponują jego różnymi definicjami, będą w stanie ustalić, która z nich jest obowiązująca,
- *autoTrim* — opcja ta służy do automatycznego usuwania danych uznanych za „stare”. Jeśli wartość pola wynosi 0, to żadne dane nie będą usuwane. Jeśli natomiast jest pozytywna, to dane starsze niż *autoTrim* (w sensie wartości stempla czasowego) od aktualnego najnowszego wpisu będą oznaczane jako dostępne do skasowania. Sterownik przechowywania danych nie jest zobligowany do ich natychmiastowego usunięcia, a system może zwracać takie „stare” dane.

- *recordSize* — rozmiar pola danych rekordu, w bajtach. Ta wartość określa jak długa jest część „danych” rekordu. Do zasadniczego rozmiaru rekordu należy dodać jeszcze 8 bajtów na stempel czasowy. Rekordy w ramach danego ciągu muszą mieć tę samą długość — co jest typowe w przypadku systemów pomiarowych, gdzie pomiar zawsze zamyka się w konkretnej liczbie bajtów,
- *options* — opcje przekazywane sterownikowi obsługującemu przechowywanie danych na dysku (patrz rozdział 4.5),
- *tombstonedOn* — informacja o tym czy (i kiedy) ciąg został skasowany. Jeśli wynosi 0, to ciąg nie został skasowany. W przeciwnym razie jest tutaj stempel czasowy, kiedy wydano polecenie skasowania ciągu. Jeśli wartość tego pola jest niezerowa, to węzeł nie dysponuje danymi danego ciągu, więc na żądanie pobrania danych czy dopisania rekordu zwróci informację o braku tegoż ciągu.

Jeśli ciąg nie jest skasowany, to istnieją także jego dane. Konkretny format zapisu tych danych zależy od danego sterownika LFD (patrz p. 4.5), nie mniej jednak zawsze na on charakter uporządkowanego ciągu rekordów. Ciąg ten może być kompletny, lub nie (jeśli węzeł uległ wcześniej awarii). Zdekompletowany ciąg do którego dopisuje się dane będzie dysponował też jednym lub więcej ciągami pomocniczymi, przechowującymi tymczasowo zapisy, dopóki „główny” ciąg nie zostanie uzupełniony brakującymi danymi z innych węzłów, pobranymi w procesie naprawczym. Dokładny opis procesu naprawczego podano w rozdziale 4.6.

4. Opis modułów programu bazodanowego

Wykonana baza danych, jako program o średnim rozmiarze²⁶ wymagał modularyzacji. Jest to podejście standardowe w przypadku projektów informatycznych i zapewnia ważne zalety, takie jak uproszczenie wymiany informacji między podzespołami czy uproszczona konserwacja i opis działania programu. Zastosowano zarówno moduły zewnętrzne, jak i moduły stworzone na potrzeby bazy. W rozdziale podano szczegółowy, techniczny opis modułów bazy oraz specyfikację protokołu sieciowego którym posługuje się baza.

4.1. *startup* — uruchomienie bazy i konfiguracja

Zadaniem modułu *startup* jest wczytanie plików konfiguracyjnych, oraz uruchomienie wątków składowych węzła. Zakłada się, że plik konfiguracyjny znajduje się w katalogu uruchomieniowym programu i nazwa się *config.json* — założenie to można zrewidować zmieniając ścieżkę w kodzie programu. Uruchamia on wątki składowe - takie jak wątek naprawczy, wątek *gossip* oraz wątek połączeń TCP, a także, jeśli zostanie to uznane za konieczne, inicjuje pobranie informacji o klastrze z wskazanego węzła.

4.2. *gcollector* — odśmiecacz metadanych

Jak wspomniano w rozdziale 3.4, metadane skasowanego ciągu, w przeciwieństwie do jego danych, nie są kasowane od razu po otrzymaniu takiego rozkazu, a trzymane przez tzw. czas bezpieczny. Zadaniem *gcollector* jest periodyczne skanowanie bazy metadanych ciągów zdefiniowanych w danym węźle oraz fizyczne usuwanie ich, jeśli są one oznaczone jako skasowane oraz wystarczająco stare.

Dokonywane jest to poprzez uruchomienie przy starcie bazy osobnego wątko odśmiecacza. Co czas bezpieczny skanuje on całą bazę metadanych. Żeby dane te zostały fizycznie skasowane, muszą być spełnione dwa warunki — ciąg musi być oznaczony jako skasowany (znacznik *tombstone*)

²⁶2845 linijek kodu

oraz od czasu oznaczenia musi minąć przynajmniej czas bezpieczny. Jeśli te warunki są spełnione, za pomocą menedżera danych definicja ciągu jest fizycznie usuwana.

Ideą ciągu bezpiecznego jest to, że awaria węzła nigdy nie będzie trwała dłużej niż ten czas (jeśli tak, to należy usunąć wszystkie dane z węzła przed włączeniem go ponownie), tak więc będzie to czas przez który „skasowane” metadane będą istniały w systemie. Po tym czasie, niezależnie od stanu awarii węzłów, za które były te ciągi odpowiedzialne, metadane nie będą już obecne w klastrze.

4.3. gossip — komunikacja stanu węzłów

Projektowana baza danych, jak system nie dysponujący węzłem typu master, musi dysponować konkretnym mechanizmem utrzymywania informacji o stanie klastra, jego składowych oraz ich parametrach. Do tego celu służy mechanizm *gossip*. Prowadzi on także wewnętrzną bazę danych stanu i parametrów innych węzłów.

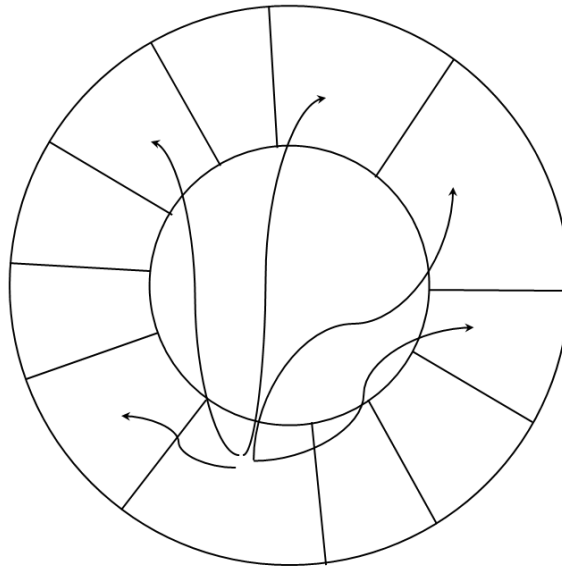
Baza danych węzłów jest elementem niezbędnym do funkcjonowania reszty systemu. Pozwala ona przede wszystkim na podstawie nazwy ciągu i numeru repliki określić węzeł zań odpowiedzialny, a także stwierdzić, że węzeł ten uległ awarii. Po otrzymaniu od innego węzła — spodziewanej lub nie — informacji o stanie klastra (bądź jego wycinka), *gossip* sprawdza czy dane te są nowsze (każda informacja bowiem opatrzona jest stemplem czasowym) i jeśli są, wprowadza je do stanu wiedzy węzła. *Gossip* służy także detekcji błędów. Gdy z jakiegoś powodu komunikacja z innym węzłem zawiedzie, zwiększany jest licznik awarii komunikacji z danym węzłem. Jeśli komunikacja powiedzie się, licznik ten jest resetowany. Dopiero gdy przekroczy on pewną wartość, docelowy węzeł oznaczony jest jako niesprawny, a informacja ta przesyłana jest innym węzłom. Nie można wyciągnąć wniosku o awarii konkretnego węzła, jeśli błąd wystąpi tylko raz, albowiem może to wynikać z szeregu innych przyczyn (chwilowe przeciążenie, przejściowa awaria sieci), więc każdy błąd musi być potwierdzony kilkukrotną obserwacją.

Oczywiście gdy węzeł ma do przekazania innym dane, nie rozsyła ich do wszystkich węzłów. Takie podejście podnosiłoby złożoność komunikacyjną systemu do $O(n^2)$, wymagając pełnej siatki topologicznej. W przypadku większych klastrów, sieć dosłownie zalewana byłaby komunikacją. W takiej sytuacji wymagane jest inteligentne rozwiązanie.

Rozwiązanie to zapobiega wysyceniu sieci w sposób dwufazowy. Jeśli węzeł odbierze informacje co do których stwierdzi że są nowe, dopisze je do swojej kolejki nowych informacji. Komunikacja nie jest więc bezpośrednio inicjowana poprzez odebranie nowych danych. Co 4 sekundy wątek *gossip* sprawdza czy są nowe informacje które powinny być rozesłane. Jeśli tak jest, wylosuje z swojej listy węzłów 5 pozycji do których spróbuje przesłać te nowe dane. Dodatkowo, jeśli nie wylosował swojego bezpośredniego sąsiada po prawej stronie, doda go do tej listy, końcowo będzie to 5 lub 6 pozycji. Mechanizm ten ilustruje rysunek 19. To obostrzenie zapewnia że w końcu wszystkie węzły otrzymają zaktualizowaną informację, choćby miało się to stać w czasie $O(n)$. Ze względu na losowanie i późniejsze rozsyłanie wiadomości do 5 węzłów przez każdy następny, stanie się to zazwyczaj dużo szybciej.

Jeśli zaś *gossip* nie ma żadnej nowej informacji do przetransmitowania, wybierze on 2 węzły (również w razie potrzeby dodatkowo doliczając prawego sąsiada) co do których sprawdzi czy jest w stanie się z nimi skomunikować. Jeśli próba zawiedzie, zostanie zwiększony lokalny licznik awarii tych węzłów — przy odpowiedniej liczbie powtórzeń awarii reszcie klastra zostanie przesłana informacja o awarii węzła, z którym komunikacja wielokrotnie zawiodła.

Ciekawym problemem jest budowanie bazy na początku rozruchu klastra. Otóż mając szereg węzłów bez informacji startowej o sobie nawzajem niemożliwe jest zbudowanie takiej bazy. Wymagane jest więc, aby węzeł który zostaje uruchomiony miał jakąś możliwość nawiązać kontakt przynajmniej z jednym węzłem. Ten problem rozwiązuje instytucja tzw. węzła startowego. Podczas rozruchu na podstawie pliku konfiguracyjnego węzeł podejmuje decyzję, czy wystosować do zdalnego węzła opisanego w swoim



Rysunek 19: Ilustracja rozsyłania danych o węzłach za pomocą *gossip*. Węzeł źródłowy wysyła komunikat do 5 wylosowanych węzłów, w których zawiera się również jego bezpośredni prawy sąsiad. To, czy każdy z docelowych węzłów podobnie retransmituje informacje zależy od tego czy wykorzysta ją do aktualizacji stanu swojej wiedzy.

pliku prośbę o przesłanie całej swojej mapy danych oraz informację o sobie, tak aby węzeł startowy mógł poinformować resztę klastra. Oczywiście węzeł startowy przy rozruchu może nie nawiązywać połączenia z żadnym innym węzłem, wszak to inne węzły skomunikują się z nim. Możliwa jest także konfiguracja gdzie dwa węzły startowe komunikują się przy starcie ze sobą, a jedna część klastra rozpoczyna inicjalizację od jednego z nich, a druga od drugiego. Węzeł startowy w żaden sposób nie jest uprzywilejowany, a służy jedynie do wstępnej propagacji danych o klastrze.

W takiej sytuacji może być jednak ciekawe zjawisko. Przypuśćmy że klaster z jednym węzłem startowym uruchomił się pomyślnie. Następnie dochodzi do awarii węzła startowego, po czym zostaje on zrestartowany. Problem jest teraz tylko tego typu, że węzeł startowy nie dysponuje żadną informacją na temat innych węzłów! Rozwiązanie tego problemu wymaga jednego dodatkowego założenia przy komunikacji *gossipa*. Zasadę tą można sformułować następująco: „jeśli skomunikuje się z tobą węzeł o adresie którego nie masz w swojej bazie węzłów, to zażądaj od niego

wysłania swojej bazy wszystkich węzłów“. Ponieważ system utrzymuje zachowanie sprawdzania komunikacji z innymi węzłami — nawet tymi, co do których stwierdzono awarię — prędzej czy później powrót węzła zostanie wykryty, a jego stan wiedzy szybko zaktualizowany.

4.4. netdispatch — multiplekser połączeń TCP

Celem uproszczenia konfiguracji sieciowej systemu zdecydowano się na ograniczenie liczby serwerowych gniazdek TCP wykorzystywanych przez serwer. Z tego powodu do wszelkiej komunikacji sieciowej wykorzystywane jest tylko jedno gniazdo TCP. Rodzajów komunikacji natomiast jest kilka, wyróżnić można chociaż transfer danych ciągów czy komunikację typu gossip. Z tego też powodu moduł netdispatch służy do multipleksowania odbieranych połączeń do różnych modułów. Typ połączenia określany jest przez pierwszy odebrany przezeń bajt, następnie połączenie przekazywane jest do odpowiedniego, innego modułu.

4.5. lfds — przechowywanie danych ciągu

Jednym z kluczowych aspektów projektowanej bazy danych jest możliwość przechowywania ich na dysku. Taką możliwość zapewnia właśnie podsystem *LFD* (*Local Filesystem Driver*). Przede wszystkim należy zacząć od opisanie jaką funkcjonalność powinien zapewniać taki podsystem. Celem modularyzacji systemu oraz możliwości stosowania różnych sposobów zapisu, nawet w obrębie jednego klastra. Zdecydowano się na definicję *LFD* jako interfejsów języka Java oraz stworzenie jednej, prostej ale skutecznej, implementacji tego zestawu interfejsów.

Zastosowanie tutaj interfejsów zamiast konkretnej implementacji pozwala dostosowywać w szerszy sposób bazę danych do swoich potrzeb. Jeśli potrzebny byłby inny schemat fizycznego zapisu niż napisany jako referencyjny, można zawsze stworzyć własną implementację interfejsów *LFD*, korzystając chociażby z relacyjnej bazy danych, lub nawet z innego systemu przechowywania danych pomiarowych.

Przed wszystkim podstawową jednostką w tym przypadku będzie ciąg

LFD, opisywany przez nazwę, rozmiar pola danych rekordu oraz dodatkowe opcje, których implementacja zależna będzie od implementacji sterownika *LFD*. Od ciągu przede wszystkim oczekuje się możliwości odczytu i dopisywania danych — na razie nie martwiąc się o replikację czy metadane. Ze względu na prostotę nie przewiduje się tutaj możliwości modyfikowania istniejących danych. Dopuszcza się jedynie dopisywanie nowych rekordów, o stemplu czasowym większym od aktualnego maksimum (tzw. głowy ciągu). Musi też istnieć opcja kasowania takich ciągów. Mając więc pewne założenia co do obiektów na których wykonywane są operacje, dalej nazywanych ciągami *LFD*, można przystąpić do zbudowania implementacji takowych.

Referencyjna implementacja, której nadano nazwę *SUZIE*, do przechowywania danych ciągu *LFD* wykorzystuje płaskie pliki, zawierające zapisane w uporządkowany sposób (rosnąco po stemplu czasowym) rekordy. Aby zmniejszyć czas dostępu do danych sterownik ten pozwala na dzielenie danych na pliki, rozmiar których można kontrolować za pomocą wspomnianych wcześniej opcji. Pliki te znajdują się w jednym katalogu, noszącym nazwę ciągu, tak więc sterownik ten wprowadza dodatkowe obostrzenie na nazwę ciągu, mianowicie by była ona poprawną nazwą katalogu w systemie plików na którym instalowana jest baza.

Pliki w których znajdują się dane ciągu nazywane są od stempla czasowego pierwszego rekordu, który się w nich zawiera. W ten sposób dysponując listą wszystkich plików można ustalić w którym pliku znajdzie się dany rekord. Dodatkowo, pliki których zapisywanie zostało definitywnie zakończone, co poznać można po utworzeniu pliku o nazwie „większej“ (w sensie numeracji) będą już otwierane tylko i wyłącznie w trybie do odczytu. Można to wykorzystać do zaaplikowania im optymalizacji na poziomie systemu plików (np. kompresja w NTFS). Wyjątkiem tutaj jest pierwszy plik danych — tworzony w momencie inicjalizacji ciągu — którego nazwa ustawiana jest na 0.

Referencyjny sterownik *LFD*, obsługując dzielenie danych ciągu na wiele plików, musi dysponować mechanizmem który zdecyduje w której chwili plik do którego aktualnie prowadzony był zapis powinien być za-

```

// Locate the starting block
int start_index = 0;
while (files.get(start_index) <= from) {
    if (start_index == files.size()-1)
        break;
    // We cannot advance further. This index is the seeked-for
    // position

    if (files.get(start_index+1) <= from) start_index++;
    else break;
}
// start_index is the index in files() of the block containing
// our starting datapoint. Locate the ending block
int end_index = start_index;
while (end_index < files.size()-1) {
    // while we can do any advancing...
    if (files.get(end_index+1) <= to) end_index++;
    else break;
}

```

Listing 2: Kod odpowiedzialny za ustalenie nazw plików zawierających podzbiór danych dotyczących zapytania.

mknięty, zaś do zapisu nowych rekordów stworzony nowy. Zachowanie sterownika *SUZIE* można kontrolować, podając mu odpowiednie opcje. Możliwe jest to przy definicji (bądź późniejszej zmianie definicji) ciągu, w polu *options* (opisane w rozdziale 3.6), podać można liczbę rekordów, po zapisaniu których zostanie stworzony nowy plik. Jeśli wartość ta nie zostanie podana, *SUZIE* obliczy taką wartość żeby wynikowe pliki były nie większe niż 16 MB.

Wyszukiwanie danych w ciągu *SUZIE* odbywa się dwufazowo. Pierwszym krokiem jest ustalenie listy plików które będą zawierać interesujące nas dane. Ponieważ zapytanie zawsze formułowane jest przy użyciu dwóch parametrów — *od* oraz *do*, będzie to z pewnością ciągły zakres. Lista plików składowych ciągu jest sortowana rosnąco, a odpowiedni zakres wyszukiwany, co ilustruje listing 2

Następnie, na podstawie stworzonej listy plików, zwracany jest obiekt wyniku. Ponieważ gwarantowane jest, że dopóki przynajmniej jeden obiekt wyniku jest w użyciu (o co dba mechanizm zliczania referencji obiektu ciągu *SUZIE*) pliki nie będą kasowane, nie ma potrzeby późniejszego

sprawdzania ich istnienia.

W obrębie plików ustalenie zakresu z którego będą pobierane dane sprowadza się do czterech możliwości:

- od *i* do wypadają poza zakresami pliku, przeczytany liniowo zostanie cały plik,
- od *i* do wypadają wewnątrz pliku, zostanie przeczytany tylko jeden plik, w całości lub w swoim podzakresie,
- od wypadła w pliku, do poza nim — ten plik zostanie przeczytany jako pierwszy, od pewnego miejsca do końca,
- od wypadła poza plikiem, do wypadła w pliku — ten plik zostanie przeczytany jako ostatni, od początku do pewnego miejsca wewnątrz tego pliku.

Odnalezienie rekordu, dzięki faktowi, że są one uporządkowane po rosnących stemplach czasowych, jest zaskakująco szybkie. Wykorzystuje się tutaj mechanizm przeszukiwania binarnego, którego implementację umieszczono na listingu 3.

Ze względu na elementarnie prosty format plików odczyty danych mogą przebiegać w sposób liniowy, najczęściej wiele rekordów na jedno systemowe wywołanie odczytu, do buforów w pamięci, bez analizy i dodatkowego przetwarzania każdego rekordu, co zapewnia dużą szybkość sterownika *SUZIE*.

Pewnego zmartwienia nastroczać może synchronizacja czytelnik-pisarz. Problem ten rozwiązano w maksymalny sposób upraszczając format plików. Upewniając się że jedyną dozwoloną formą modyfikacji plików jest dopisywanie nowych rekordów, efektywnie zlikwidowano wpływ modyfikacji pliku na odczyt z niego w danej chwili.

Dodatkowym wyzwaniem które sterownik *LFD* musi obsługiwać, jest funkcja *autoTrim*, opisana w rozdziale 3.6. Ze względu na zakaz modyfikacji plików w inny sposób jak tylko dopisywanie danych, jednostką kasowania danych staje się tutaj cały plik. Dopiero, gdy warunek że

```

while (imax >= imin) {
    imid = imin + ((imax - imin) / 2);
    this.cfile.position(imid*(8+this.recsize));
    this.cfile.read(this.timestamp_buffer);
    this.timestamp_buffer.flip();
    temp = this.timestamp_buffer.getLong();
    this.timestamp_buffer.clear();
    if (temp == time)
        return imid;
    if (temp < time) {
        if (imid == amax)
            // Range exhausted right-side
            if (is_start)
                throw new IllegalArgumentException();
            else
                return amax;
        imin = imid + 1;
    } else {
        if (imid == 0)
            // Range exhausted left-side
            if (is_start)
                return 0;
            else
                throw new IllegalArgumentException();
        imax = imid - 1;
    }
}
// Still not found. Interpolate.
if (is_start) {
    if (temp < time)
        return imid+1;
    else
        return imid;
} else {
    if (temp < time)
        return imid;
    else
        return imid-1;
}

```

Listing 3: Wycinek funkcji przeszukiwania binarnego.

wszystkie obiekty wyników są zamknięte można przystąpić do kasowania plików. Wtedy też uruchamiana jest procedura *autoTrim*, jeśli jest wymagana. Dopiero gdy *SUZIE* wykryje że plik zawiera tylko dane, które w całości mieszczą się za poza obrębem danych które mają być dostępne, jest on kasowany. W zależności od wybranego poprzez opcje rozmiaru plików może to powodować że nawet duża ilość danych archiwalnych będzie dostępna do odczytu, nawet mimo bardzo restrykcyjnego ustawienia *autoTrim* na dany ciąg. Uruchamiając tą operację dopiero gdy wszystkie obiekty wyniku są zamknięte i blokowanie zwracania następnych obiektów tego typu do czasu ukończenia operacji zapewnia się spójność danych i eliminuje hazardy.

4.6. store — menedżer danych i metadanych ciągów

Store jest mechanizmem łączącym przechowywanie danych ciągu oraz jego metadanych. Pośredniczy żądaniom odczytu i zapisu danych, oraz odpowiada za odpowiednią ich synchronizację, wraz z tworzeniem i modyfikowaniem ciągów. Sam *store* składa się z trzech zasadniczych części — *SeriesDB*, *SeriesController* i *WriteAheadContext*.

SeriesDB jest w ramach węzła singletonem. Stanowi podstawowy punkt wejścia do procesów operujących na ciągach oraz ich metadanych, zapewniając w ramach swoich działań synchronizację. Zajmuje się redefinicją metadanych ciągów oraz tworzeniem obiektów *SeriesController*.

SeriesController jest klasą reprezentującą konkretny ciąg, zarówno w kontekście jego sterownika LFD i danych, jak i metadanych. Wywołania *store* zapewniają, że dla danego ciągu w danej chwili będzie istniał maksymalnie jeden *SeriesController*, dzięki temu obiekt ten może wewnętrznie przeprowadzać synchronizację bez konieczności koordynowania jej z innymi obiektami, co pozwala w istotny sposób uprościć implementację tego zagadnienia.

WriteAheadContext jest klasą odpowiedzialną za pewne składowe procesu naprawczego. Ponieważ, jak wspomniano wcześniej, do ciągu LFD możliwe jest tylko dopisywanie, a bazie nie wolno odmówić zapisu ze

względem na zdekompletowany ciąg, wymagane jest miejsce na tymczasowe przechowywanie takich „zawieszonych“ rozkazów — tych, które zostały przez bazę przyjęte, ale jeszcze nie dopisane do odpowiedniego zasadniczego ciągu LFD. Obsługę tej funkcjonalności zawiera właśnie *WriteAheadContext*, tworzony dla konkretnego ciągu.

Jak już wcześniej wspomniano, *SeriesDB* ma dwa zadania. Pierwszym z nich jest tworzenie i zarządzanie obiektami *SeriesController*. Ponieważ z założenia musi istnieć tylko jeden taki obiekt na otwarty ciąg, *SeriesDB* prowadzi także pamięć podręczną takich obiektów i zarządza nimi za pomocą zliczania referencji, dzięki czemu pozostałe elementy systemu korzystające z ciągów, zwłaszcza interfejs klienta, mogą operować w przejrzysty sposób. Drugim zasadniczym zadaniem *SeriesDB* jest (re)definicja ciągów. Ponieważ z obiektem *SeriesController* związana jest konkretna definicja ciągu, nie może ona zostać zmieniona w momencie gdy taki obiekt jest otwarty. Utrzymywana jest kolejka zadań redefinicji. Uruchamiane są one dla danego ciągu w momencie kiedy zostanie wykryte że jego obiekt *SeriesController* nie znajduje się w użyciu. Dodatkowo nowy *SeriesController* nie zostanie utworzony gdy redefinicja nie została ukończona. Dzięki tej operacji zachowuje się gwarancję synchronizacji.

Oczywiście usuwanie ciągu z pamięci w momencie kiedy nie jest on już używany może wydawać się dobrym pomysłem, jednak jest to zachowanie szkodliwe z punktu widzenia długoterminowej wydajności. *SeriesDB* implementuje więc pamięć podręczną. Rozmiar jej można ustawić w pliku konfiguracyjnym. Stwierdzono że logika LRU²⁷, polegająca na zliczaniu częstotliwości użycia danego ciągu i w przypadku przekroczenia rozmiaru pamięci podręcznej eksmisja z niej najrzadziej wykorzystywanych ciągów, nie będzie pomocna w przypadku użycia gdy wszystkie (lub większość) ciągów są okresowo wykorzystywane w celu dopisania wartości. Z tego powodu też zdecydowano się na odmienne, prostsze podejście. W momencie przekroczenia rozmiaru pamięci podręcznej usuwane są z niej wszystkie ciągi, które aktualnie nie znajdują się w użyciu. Testy prowadzone w trak-

²⁷Least Recently Used

cie tworzenia bazy dowiodły że zysk polegający na implementacji pamięci podręcznej w ten sposób w porównaniu do usuwania ciągów z pamięci w momencie ich zamknięcia uzasadnia jej użycie.

Przechowywanie metadanych przez *SeriesDB* zrealizowane jest w prosty sposób. W katalogu, określonym w pliku konfiguracyjnym, znajdują się pliki o nazwach odpowiadających nazwom ciągu. Pliki te zawierają zserializowane (za pomocą mechanizmów serializacji języka Java) definicje metadanych. Zdecydowano się na serializację Java, gdyż nie ma wymogu co do czytelności, czy też możliwości edycji "przez człowieka" tych danych, a serializacja jest jednym z szybszych sposobów wczytywania tych obiektów. Prędkość istotnie jest zagadnieniem istotnym, gdyż metadane wczytywane są za każdym razem kiedy otwierany jest ciąg który nie znajduje się w pamięci podręcznej *SeriesDB*.

Obiekt *SeriesController* reprezentuje konkretny ciąg, posiadający ustaloną definicję. Definicja ta pozostaje niezmienna w czasie życia obiektu. Stanowi on podstawową bramę do przeprowadzania operacji na ciągu, takich jak odczyt danych, dopisanie rekordu czy chociażby pobranie ostatnio dopisanego stempla czasowego. Zawiera on również implementację operacji skasowania danych ciągu (nie metadanych, gdyż jest to zadaniem *SeriesDB*). Najistotniejszym jednak działaniem *SeriesController* jest branie udziału w wykrywaniu zdekompletowania ciągu i współpraca z *WriteAheadContext* oraz modulem naprawczym w celu „uzdrowienia” ciągu. Każde polecenie zapisu oprócz kompletnego rekordu zawiera również stempel czasowy ostatnio zapisanego. W momencie gdy wartość ta nie zgadza się z rekordem zapisanym w bazie, uruchamiany jest proces naprawczy. Zapis przejmuje *WriteAheadContext*, oraz informowany o awarii jest wątek naprawczy. W momencie gdy uzupełnione zostaną brakujące dane, jest o tym informowany *WriteAheadContext* który scala zebrane dane w spójną całość. *SeriesController* odpowiedzialny jest również za realizację opcji *autoTrim*, wysyłając sterownikowi *LFD* rozkazy co do „ucinania” zbioru danych. Operację zapisu widzianą z poziomu *SeriesController* przedstawiono na listingu 4.

```

/**
 * Adds a record
 * @param previousTimestamp timestamp of previous write
 * @param currentTimestamp timestamp of this write
 * @param value value to write to DB
 * @throws IllegalArgumentException value passed was malformed
 */
public synchronized void write(long previousTimestamp,
                                long currentTimestamp,
                                byte[] value)
    throws IllegalArgumentException, IOException {

    long rootserTimestamp = this.primary_storage
                            .getHeadTimestamp();
    if (currentTimestamp <= rootserTimestamp)
        return; // just ignore this write

    if (previousTimestamp == rootserTimestamp) {
        // this is a standard LFD-serializable
        this.primary_storage.write(currentTimestamp,
                                    value);
        this.wacon.signalWrite(currentTimestamp);

        if (this.series.autoTrim > 0)
            this.primary_storage.trim(
                currentTimestamp - this.series.autoTrim);
    } else {
        this.wacon.write(previousTimestamp, currentTimestamp,
                          value);

        // Now we need to signal that this series needs a repair...
        ReparatorySupervisorThread.getInstance().postRequest(
            new RepairRequest(this.series,
                              this.primary_storage
                                  .getHeadTimestamp(),
                              previousTimestamp));
    }
}

```

Listing 4: Operacja dopisania rekordu w *SeriesController*.

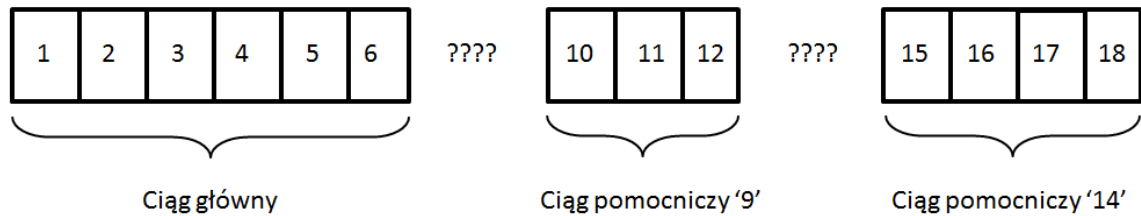
Istotną częścią organizacji danych podczas naprawy zajmuje się podległa *SeriesController* instancja klasy *WriteAheadContext*. Jak już wcześniej wspomniano, ze względu na fakt że wykorzystywany interfejs *LFD* pozwala jedynie na dopisywanie nowych rekordów i odczyt zakresów, wymagana jest tymczasowa pamięć na przechowanie zapisów podczas naprawy ciągu. Nie można bowiem dopisać do zasadniczego ciągu *LFD* nowych wartości, gdy „w środku” znajdowałaby się luka. Zadanie przetrzymania tych nowo dodawanych danych podczas gdy ciąg nie jest jeszcze naprawiony zajmuje się właśnie *WriteAheadContext*.

Do zasadniczego ciągu przypisanych może być dodatkowo kilka ciągów dodatkowych, służących do przechowywania takich danych. W najprostszym przypadku pojawi się jeden taki ciąg. Rozważmy przypadek gdy węzeł, ze względu na awarię, przeoczył pewien ciąg zapisów. W momencie gdy pojawi się rozkaz zapisu z „poprzednim stemplem czasowym”, tworzony jest ciąg pomocniczy o nazwie równej wartości „poprzedniego stempla czasowego”. Będzie on miejscem gdzie wędrować będą nowe zapisy. Równocześnie zostanie uruchomiony proces naprawczy, który po pewnym czasie uzyskawszy brakujące dane zapisze je do głównego ciągu. Gdy dojdzie już do elementu który stanowił „poprzedni stempel czasowy”, ciąg pomocniczy zostanie w całości dopisany do ciągu głównego. Następnie ciąg pomocniczy zostanie usunięty, zaś nowe zapisy będą od razu dodawane do ciągu głównego.

Oczywiście może wydarzyć się sytuacja w której węzeł ponownie ulegnie awarii, nawet nie zdążywszy naprawić ciągu. W takim razie pojawi się kolejna luka w systemie. Kolejne zapisy do ciągu umieszczone będą już nie w pierwszym ciągu pomocniczym, gdyż logicznie po tym ciągu były jakieś zapisy. Stworzony zostanie nowy ciąg pomocniczy, noszący nazwę kolejnego „poprzedniego stempla czasowego”. Taką hipotetyczną sytuację ilustruje rysunek 20.

Istotnym zastrzeżeniem jest również to, że operacja zapisu przedstawiana przez *SeriesController* służy zarówno do dopisywania danych otrzymanych od użytkownika, jak i do przeprowadzania procesu naprawczego.

W przypadku takiej sytuacji *WriteAheadContext* musi poprawnie zdecydować do którego ciągu konkretny element powinien być dopisany, oraz — w razie czego — połączyć ciągi w jeden.



Rysunek 20: Graficzna reprezentacja możliwego stanu ciągu po dwóch awariach. Pierwsza awaria nastąpiła po zapisie rekordu 6. Kolejny zapis (o poprzednim stemple 9) poskutkował stworzeniem ciągu pomocnicznego '9'. Kolejna awaria wystąpiła zanim ciąg mógł zostać naprawiony i węzeł otrzymał dopiero rozkaz zapisu rekordu '15' z poprzednim stemplem równym '14'. Utworzony został więc pomocniczy ciąg '14'. Ciągi te będą reintegrowane z głównym ciągiem gdy moduł naprawczy wyda polecenie zapisu rekordu o stemple równym nazwie ciągu pomocniczego.

WriteAheadContext odpowiedzialny jest również za dopisanie ciągów pomocniczych do ciągu głównego gdy dostępne są już brakujące dane. Ciągi pomocnicze również korzystają z interfejsów *LFD*, co upraszcza konstrukcję systemu. Może to hipotetycznie posłużyć do korzystania z dwóch różnych sterowników *LFD* — jednego do przetrzymywania ciągów pomocniczych, a drugiego do głównego (z definicji niemalże dłuższego i ważniejszego) ciągu danych.

4.7. repair — naprawa uszkodzonych ciągów

Mimo obecności podsytému danych służących do radzenia sobie z sytuacjami awaryjnymi, musi istnieć pewien moduł który będzie reagował na zgłoszone uszkodzenia ciągów i dokonywał napraw. Taką funkcjonalność implementuje moduł *repair*.

Podstawowym punktem wejścia w module jest wątek nadzorcy naprawy. Wątek ten otrzymuje informacje od *SeriesController* o konieczności inicjacji naprawy danego ciągu. Jeśli przez czas ok. 24 sekund nie otrzyma żadnych

informacji, dokona on próby przeskanowania katalogu z ciągami pomocniczymi. Mogła bowiem nastąpić sytuacja w której wykryta została awaria ciągu, ale węzeł ponownie został wyłączony. W tym wypadku pozostanie ślad w postaci utworzonego ciągu pomocnicznego, ale *SeriesController* wykryje awarie dopiero po otrzymaniu kolejnego żądania zapisu.

W momencie identyfikacji potrzeby naprawy ciągu, wątek-nadzorca sprawdzi czy nie jest to duplikat poprzednio złożonego zgłoszenia i w razie potrzeby wyeliminuje go. Zostanie również sprawdzone czy nie jest to zgłoszenie stare i już realizowane, a ciąg nie został chociażby w tym czasie skasowany. Jeśli wszystkie warunki początkowe do naprawy zachodzą, kontrola przekazywana jest do zasadniczej części modułu naprawczego.

Zadaniem tej części jest uzyskanie brakujących danych i przekazanie ich do odpowiedniego obiektu *SeriesController*. Czynione jest to poprzez odpytanie węzłów odpowiedzialnych za inne repliki. W razie awarii innego węzła odpowiedzialnego za replikę, wypróbowywany jest kolejny węzeł, do skutku. Jeśli w danej chwili żaden węzeł z brakującymi danymi nie jest dostępny, proces jest zatrzymywany. Jedyne ślad który w tej sytuacji pozostanie to ciąg pomocniczy, tak więc wątek-nadzorca w końcu ponowi tą naprawę.

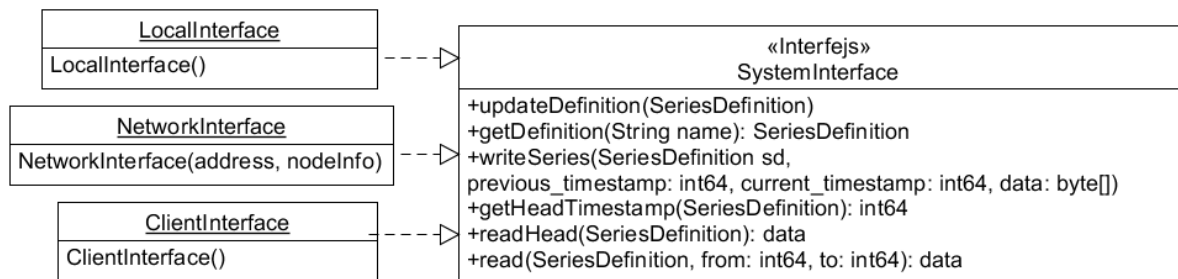
Jeśli jednak udaje się połączyć z węzłem i pobrać oraz zapisać brakujące dane, kończy się tutaj działanie naprawy ze strony modułu *repair*. Resztę działań wykona *WriteAheadContext*, zapisując fizycznie dane oraz łącząc ciąg pomocniczy z ciągiem głównym.

4.8. ifc — interfejs użytkownika i międzywęzłowy

Ifc jest mechanizmem który zapewnia łączność międzywęzłową jeśli chodzi o operacje i przesył danych, oraz możliwość zrealizowania żądań klienta.

W najprostszej swojej postaci baza danych udostępnia szereg funkcji. Są to podstawowe operacje które klient może wykonać bezpośrednio na bazie, takie jak zapis, odczyt czy stworzenie nowego ciągu. Ze względu na wybrany przez klienta interfejs funkcje te mogą być realizowane na

różne sposoby. Funkcje te, wraz z sposobami wykonania, przedstawiono na rysunku 21.



Rysunek 21: Diagram UML interfejsu wraz z trzema istniejącymi klasami implementującymi go.

LocalInterface jest implementacją która operacje wykonuje przy użyciu tylko i wyłącznie lokalnych zasobów, tj. korzystając z funkcjonalności *SeriesDB*. Jeśli jakiś ciąg nie jest dostępny na węźle, zwróci on taką informację. Jest więc mechanizmem dostępu do operacji udostępnianych przez bazę lokalnie.

ClientInterface wykonuje operacje biorąc pod uwagę cały klaster. W przypadku gdy nie jest on odpowiedzialny za dany węzeł, połączy się z innym by zrealizować tą operację. Wszelkie operacje redefinicji są przesyłane do węzłów odpowiedzialnych.

NetworkInterface jest zdalną reprezentacją interfejsu jakiegoś węzła. Nie wykonuje on operacji poza połączeniem się z innym węzłem i zlecenia wykonania mu jakiejś operacji.

Kolejnym, bardzo istotnym elementem modułu *ifc* jest klasa *Network-CallInbound*. Przetwarza ona komunikację sieciową na gniazdku na wywołania interfejsu. To, którego interfejsu uwzględnionego na rysunku 21 dotyczy zapytanie, zależy od charakteru połączenia sieciowego z *ifc*. Istnieją dwie kategorie połączeń — międzywęzłowych oraz klienckich. Po drugiej stronie połączenia klienckiego znajduje się moduł którego zadaniem jest zapisanie bądź odczytanie danych, bez wnikania w strukturę klastra. Zostanie on obsłużony więc przy pomocy implementacji *ClientInterface*, który zadba o to aby jego żądanie zostało zrealizowane, choćby niektóre węzły nie działały. Do tego celu *ClientInterface* będzie musiał połączyć się z innymi

węzłami. W przypadku tego połączenia międzywęzłowego *NetworkCallInbound* korzystać będzie z implementacji *LocalInterface*. Dzieje się tak gdyż węzeł realizujący polecenia klienckie wie z którym węzłem należy się połączyć (dzięki bazie *gossip*). Dodatkowo, wymagana jest logiczna topologia pełnej siatki, tak aby każdy węzeł mógł łączyć się z dowolnym innym.

Podjęcie takie wprowadza więc dodatkową zaletę systemu — każdy z węzłów w klastrze może realizować polecenia dla klientów, i nie jest istotne to który węzeł tą operację wykona. Pozwala to na tolerancję awarii ze strony klientów. Gdy jeden węzeł zawiedzie, klienci mogą połączyć się z innym i dalej korzystać z bazy. Możliwe jest również zawężenie puli węzłów „interfejsów” chociażby przy pomocy mechanizmów sieciowych takich jak firewall czy router.

Dzięki zastosowaniu podejścia interfejsu możliwe było zastosowanie jednego kodu (tak jak i protokołu) obsługującego połączenia między węzłami jak i połączenia klienckie. Klasa *NetworkCallInbound* w momencie tworzenia jej pobiera jako parametr interfejs za pomocą którego realizować będzie zdekodowane polecenia.

Aby jednak w sposób sprawny obsługiwać polecenia *ClientInterface* musi dysponować pewną logiką i zasadami realizacji poleceń klienckich. Tak więc w przypadku odczytu danych klasa ta sprawdza czy aby to węzeł aktualnie wykonujący polecenie nie dysponuje tymi danymi. W takim wypadku nie będzie potrzeby otwierania *NetworkInterface*, a polecenie może być wykonane przy użyciu *LocalInterface*. Kod, ze względu na wykorzystanie interfejsów języka Java, pozostaje identyczny. *Gossip* dysponuje funkcją zwracającą listę węzłów odpowiedzialnych za dany ciąg z precedencją dla węzła lokalnego - tak więc jeśli będzie to węzeł lokalny, zostanie on zwrócony jako pierwszy. Klasa *InterfaceFactory* zwraca interfejs po wskaźniku na węzeł, albo *NetworkInterface* jeśli to węzeł zdalny, albo *LocalInterface* jeśli jest to węzeł który wykonuje to polecenie. Dodatkowo *InterfaceFactory* implementuje pamięć podręczną połączeń, aby nie wykonywać nawet i kilku połączeń TCP na jedno żądanie, gdyż bardzo ucierpiałyby na tym wydajność bazy.

Nieco inne obostrzenia występują w przypadku pobrania wartości maksymalnego stempla czasowego. W tym przypadku odpytywane są wszystkie odpowiedzialne węzły, możliwe bowiem, że któryś z nich zwrócił awarię, a informacja którą w tym przypadku zwrócimy klientowi może doprowadzić do awarii, ponieważ bardziej aktualne węzły zignorują te zapisy. W takiej sytuacji węzeł który uległby awarii dysponowałby różnymi danymi niż pozostałe, czyli system uległby desynchronizacji. Podobnie wygląda sytuacja zapisów, czy aktualizacji definicji ciągu. Kod realizujący w tym wypadku zapis przedstawiono na listingu 5.

W dowolnej chwili, jeśli łączność z innym węzłem w ramach *NetworkInterface* zawiedzie, informacja ta zostanie przekaza do modułu *gossip* celem śledzenia awarii węzłów.

4.9. Protokoły sieciowe

Celem skomunikowania się klienta z bazą danych oraz elementów klastra ze sobą wzajemnie, niezbędne jest zdefiniowanie protokołu sieciowego. Szczególnie istotne jest precyzyjne określenie protokołu za pomocą którego klienci będą łączyć się z bazą, wysyłać swoje polecenia i otrzymywać odpowiedzi.

Ponieważ wszystkie te rodzaje komunikacji dokonują się na jednym porcie TCP, wymagany jest jakiś sposób na odróżnienie ich po sobie. Klient sieciowy jako pierwszy bajt wysyła właśnie typ połączenia, za pomocą którego *netdispatch* przekieruje połączenie do odpowiedniego modułu. Jeśli będzie to bajt:

- 0 — jest to przychodzący komunikat *gossip*,
- 1 — jest to połączenie danych międzywęzłowe. Należy uruchomić *NetworkCallInbound* z implementacją *LocalInterface*,
- 2 — jest to połączenie od klienta bazy. Należy uruchomić *NetworkCallInbound* z implementacją *ClientInterface*.

Najprostszym wariantem jest połączenie typu *gossip*. Polega ono na wysłaniu czterobajtowego znacznika długości komunikatu, a następnie

```

public void writeSeries(SeriesDefinition serdef, long prev_timestamp,
    long cur_timestamp, byte[] data) throws LinkBrokenException,
    IllegalArgumentException, IOException, SeriesNotFoundException,
    DefinitionMismatchException {

    NodeDB.NodeInfo[] nodes = NodeDB.getInstance()
        .getResponsibleNodesWithReorder(serdef.seriesName,
                                         serdef.replicaCount);

    int successes = 0;

    for (NodeDB.NodeInfo ni : nodes) {
        if (!ni.alive) continue;

        for (int i=0; i<2; i++) { // for retries in case of failure
            SystemInterface sin = null;
            try {
                sin = InterfaceFactory.getInterface(ni);
            } catch (IOException e) {
                FailureDetector.getInstance().onFailure(ni.nodehash);
                continue;
            }
            try {
                sin.writeSeries(serdef, prev_timestamp,
                               cur_timestamp, data);
                successes++;
                break;
            } catch (LinkBrokenException | IOException e) {
                FailureDetector.getInstance().onFailure(ni.nodehash);
            } catch (DefinitionMismatchException |
                SeriesNotFoundException |
                IllegalArgumentException e) {
                throw e;
            } finally {
                sin.close();
            }
        }
    }
    if (successes == 0) throw new LinkBrokenException();
}

```

Listing 5: Operacja dopisania rekordu w *ClientInterface*.

samego komunikatu. Komunikat taki jest zserializowaną za pomocą mechanizmów języka Java klasą zawierającą opis danych i dokładnego charakteru komunikatu, co jest do przetworzenia bezpośrednio przez moduł *gossip*.

W przypadku bajtu 1 lub 2 sytuacja jest znacznie bardziej skomplikowana. Co prawda protokół, ze względu na zastosowany interfejs, w obu wypadkach pozostaje taki sam, jednak niemożliwe staje się zastosowanie mechanizmów serializacji języka Java do przesyłania komunikatów, gdyż wtedy z bazy korzystać mogliby tylko klienci napisani w tym języku programowania.

Innymi słowy, protokół ten definiowany jest bardziej przy pomocy przenośnej funkcjonalności języka Java, jaką jest *DataStream*, pozwalający zapisać zmienne w postaci bajtów, interpretowalnych przez mechanizmy również innych języków programowania.

Po pierwszym bajcie, za pomocą którego *netdispatch* ustala charakter połączenia, *NetworkCallInbound* oczekiwać będzie na kolejny bajt, opisujący rodzaj komendy. Czas oczekiwania ustalony jest na 4 sekundy, tak więc maksymalnie tak długo połączenie może być utrzymywane bez realizacji żadnej komendy. Jeśli klient nie zamierza wykonać żadnego polecenia, zaleca się aby od razu zamknął połączenie.

W dalszej części tego rozdziału zamieszczono specyfikację protokołu. Pisana jest ona przy użyciu charakterystycznych typów danych, sprecyzowanych w tabeli 3. Dodatkowy typ (definicja ciągu), pojawiający się w specyfikacji poleceń, opisano w tabeli 4.

Gdy *NetworkCallInbound* zostanie uruchomiony dla danego połączenia, odbiera bajt i parametry komendy.

Jeśli to bajt 0, to jest to polecenie pobrania definicji ciągu. Parametrem jest *string* zawierający nazwę ciągu. Następnie serwer zwróci bajt statusu. Jeśli będzie to 2, to oznacza że definicja nie występuje w systemie. Jeśli jest to 1, to wystąpił wewnętrzny błąd węzła lub klastra (należy ponowić zapytanie), a jeśli 0 to serwer odeśle do klienta definicję²⁸.

²⁸format opisano w tabeli 4

Tabela 3: Opis typów danych używanych w specyfikacji (typ zapisu *big endian*)

Nazwa	Rozmiar w bajtach	Opis
byte	1	bajt bez znaku
short	2	liczba ze znakiem w reprezentacji uzupełnień do dwóch
int	4	liczba ze znakiem w reprezentacji uzupełnień do dwóch
long	8	liczba ze znakiem w reprezentacji uzupełnień do dwóch
string	więcej niż 2	najpierw short z długością ciągu, dalej ciąg bajtów w kodzie zmodyfikowanym UTF-8.

Tabela 4: Opis definicji ciągu w reprezentacji sieciowej (pola podane w kolejności)

Nazwa	Typ	Opis
replicaCount	int	Ilość replik ciągu
recordsize	int	Rozmiar pola danych rekordu
generation	long	generacja definicji
autoTrim	long	wartość opcji <i>autoTrim</i> , opisaney w rozdziale 3.6
tombstonedOn	long	wartość znacznika skasowania ciągu , opisanego w rozdziale 3.6
options	string	opcje dla sterownika <i>LFD</i> zarządzającego ciągiem
seriesName	string	nazwa ciągu

Jeśli jest to bajt 1, to jest to polecenie zdefiniowania (bądź redefinicji ciągu). Po tym klient wysyła definicję, która ma zostać wprowadzona do systemu. Jeśli redefinicja przebiegnie pomyślnie, klient odeśle bajt 0, jeśli wystąpi błąd — bajt 1.

Jeśli to bajt 2, to jest to polecenie pobrania maksymalnego stempla czasowego. Jeśli zostanie zwrócony bajt 1 to wystąpił wewnętrzny błąd. Zwrócony bajt 2 to kod nieodnalezienia definicji ciągu. Zwrócony bajt 3 to informacja o tym że definicja którą przesłał klient, jest przestarzała (zmiana generacji). Jeśli zwrócony jest bajt 0, to następny wysłany zostanie przez serwer long z znalezionym stemplem czasu.

Jeśli to bajt 3 to jest to polecenie dopisania rekordu. Następnie klient wysyła definicję szeregu oraz znacznik czasowy poprzedniego zapisu i stempla czasowego bieżącego rekordu w formacie long (łącznie dwa znaczniki). Następnie wysyła długość pola danych w formacie short oraz pole danych rekordu. Do bazy zostanie zapisany rekord złożony z stempla czasowego bieżącego rekordu i danych. Teraz serwer odpowiada jednym bajtem statusu. Jeśli będzie to 4, to długość pola danych jest niezgodna ze specyfikacją ciągu. Jeśli jest to 3, to definicja którą przesłał klient, jest przestarzała. Jeśli ciąg nie został znaleziony, zwrócone zostanie 2, jeśli wystąpił ogólny błąd — 1. Jeśli operacja zostanie zrealizowana poprawnie, zwrócony zostanie bajt 0.

Jeśli jest to bajt 4, to jest to polecenie odczytania zakresu. Następnie klient wysyła definicję szeregu oraz dwa znaczniki zakresu odczytu, czyli znacznik „od“ i „do“ (łącznie dwa znaczniki) w formacie long. Następnie serwer odpowiada bajtem statusu, gdzie 1 to ogólny błąd, 2 to nieodnaleziony ciąg, 3 to przestarzała definicja, zaś 4 to błąd zakresu (choćby „do“ mniejsze od „od“). Zero to kod transmisji pomyślnej — wysłana zostanie następnie seria danych w postaci stempla czasowego typu long i bajtów danych — tyle ile wynosi rozmiar pola danych w tym ciągu. Transmisję tą kończy wartość -1 typu long.

Jeśli jest to bajt 5, to jest to polecenie odczytania najnowszego rekordu. Następnie klient wysyła definicję szeregu. Następnie serwer odpowiada

bajtem statusu, gdzie 1 to ogólny błąd, 2 to nieodnaleziony ciąg, 3 to przestarzała definicja. Zero to kod transmisji pomyślnej, wysłany zostanie następnie rekord w postaci stempla czasowego typu long i bajtów danych — tyle ile wynosi rozmiar pola danych w tym ciągu. Transmisję tą kończy wartość -1 typu long. Jeśli w danym ciągu nie ma ani jednego rekordu, -1 zostanie odesłane natychmiastowo.

Komunikacja międzywęzłowa przebiega na identycznych zasadach, z zastrzeżeniem takim że na te komendy węzeł odpowiada bez konsultacji z innymi węzłami, jednak to zagadnienie z punktu widzenia czysto klienckiego jest mniej interesujące.

5. Instalacja i konfiguracja systemu

W tym rozdziale ujęto zasady służące do konfiguracji klastra tak, aby pracował on w sposób stabilny i szybki. Przede wszystkim wyjaśniono wymagania sprzętowo-programowe, to w jaki sposób należy konfigurować pojedyncze węzły. Ujęto również zagadnienia dotyczące naprawy węzłów w przypadku awarii. Kolejno opisano zasady projektowania schematu bazy, czyli konkretnych ciągów. Wspomniano również o zasadach dostępu klienckiego.

5.1. Wymagania sprzętowo-programowe

Ze względu na możliwość pracy w systemach heterogenicznych baza nie wymaga konkretnego systemu operacyjnego. Minimalnym wymaganiem programowym jest to, aby system dysponował implementacją standardu Java w wersji przynajmniej 1.7. Wymagany jest również dostęp do sieci (przynajmniej jeden interfejs sieciowy), oraz miejsce dyskowe. Należy tak zaplanować dostępne miejsce na dysku, aby w całości był się w stanie zmieścić na nim największy ciąg, który ma być przechowywany w systemie, gdyż baza nie dzieli ciągów między węzły.

Konsekwentnie, należy określić katalog w systemie plików, w którym baza będzie składować dane. Przede wszystkim należy określić, na jakiej partycji znajdować się będą dane. Nie powinna być to partycja z nadmiarowym RAID, jeśli planuje się korzystać z wielu węzłów i funkcji replikacji danych.

Od strony sieci wymaga się, aby każdy z węzłów mógł nawiązać połączenie z dowolnym innym. Najprostszym sposobem realizacji jest umieszczenie wszystkich węzłów w jednej sieci lokalnej, nadając im różne adresy IP. Sieć musi zapewniać możliwość prowadzenia komunikacji typu TCP. Klient systemu bazodanowego musi móc połączyć się przynajmniej z jednym, dowolnym klientem. Zalecane jest, aby mógł połączyć się z kilkoma (na wypadek awarii jednego z węzłów dostępowych).

Sieć, w której jest instalowana baza, ze względu na uwarunkowania

bezpieczeństwa i brak wsparcia dla uwierzytelniania i szyfrowania, musi znajdować się w jednym, zaufanej podsieci IP. W przypadku systemów, które nie mogą spełnić tego warunku ze względów lokalowych, należy zastosować technologię VPN w celu stworzenia wirtualnej sieci. Obecne na rynku rozwiązania VPN pozwalają zestawić połączenie szyfrowane, więc w zapewniony jest warunek zarówno zaufania jak i pojedynczej sieci IP.

Należy zwiększyć na systemach uniksowych liczbę maksymalnych otwartych deskryptorów plików na proces, do wartości równej przynajmniej 120% maksymalnej liczby ciągów trzymanyh w pamięci cache, ponieważ na jeden otwarty ciąg przypada jeden otwarty deskryptor, zaś zapytania odczytu generują dodatkowe deskryptory.

Spełnienie warunków istotne jest zwłaszcza przy wdrożeniach lokalnych, gdzie baza występować będzie blisko systemów pomiarowych, a warunki nie zawsze są pod kontrolą administratora czy instalatora. Ze względu na pełną kontrolę środowiska, dostępną na infrastrukturze serwerowej SMOK, na której baza będzie wdrożona, warunki te są spełnione.

5.2. Planowanie klastra

Przed wszystkim rozplanowując układ klastra należy najpierw określić iloma węzłami dysponuje się, oraz jaka jest ich „moc przerobowa”. Pojemność dyskowa powinna być proporcjonalna do pamięci operacyjnej oraz mocy procesora, ze względu na to, że obciążenie tych komponentów będzie wprost proporcjonalne do pojemności.

Mając określone węzły, należy przypisać im konfigurację sieciową, tj. adres IP i port na którym mają one działać. Dany zestaw powinien być dostępny z sieci lokalnej. Port można pozostawić domyślny (tj. 8886), gdyż będzie go można zawsze dostosować w pliku konfiguracyjnym.

Kolejnym, najważniejszym w zasadzie, etapem projektowania klastra jest przypisanie węzłom przedziałów odpowiedzialności. Ponieważ węzeł dla ciągu ustala się po wyniku funkcji mieszającej, należy przyporządkować im zakresy zbioru wartości tej funkcji. Oznacza to „pokrojenie”

Tabela 5: Przykład projektu małego klastra

IP	Miejsce na dane	Początek przedziału
192.168.0.2	100 GB	-9223372036854775808
192.168.0.3	150 GB	-8627931621305730000
192.168.0.4	500 GB	-7734770997982150000
192.168.0.5	300 GB	-4757568920236910000
192.168.0.6	2 TB	7437050790207630000

między węzły zakresu liczb odpowiadającego typowi *long* języka Java²⁹. Należy zachować zasadę, że „kawałek” przestrzeni powinien być proporcjonalny do przestrzeni dyskowej. „Kawałek” oznacza się poprzez podanie wartości, od której rozpoczyna się przedział danego węzła, tak więc wartość -9223372036854775808 musi wystąpić w jednym z węzłów. Do wyznaczenia takiego schematu może posłużyć arkusz kalkulacyjny. Pewną propozycję dla przykładowych danych przedstawiono w tabeli 5.

Kolejnym krokiem jest wyznaczenie węzłów startowych, do których inne węzły będą się łączyć celem pobrania informacji o stanie sieci. Węzły takie stanowią „punkty zborne” przy inicjalizacji klastra. Zamiast wybrać jeden z nich, można też wybrać dwa albo trzy, należy jedynie upewnić się, że one same będą łączyć się z innym węzłem startowym. Taka ścieżka „węzłów startowych dla węzłów startowych” powinna zawierać cykl, ze względu na możliwe awarie węzłów podczas startu. Przykład takiego rozplanowania przedstawiono na rysunku 22.

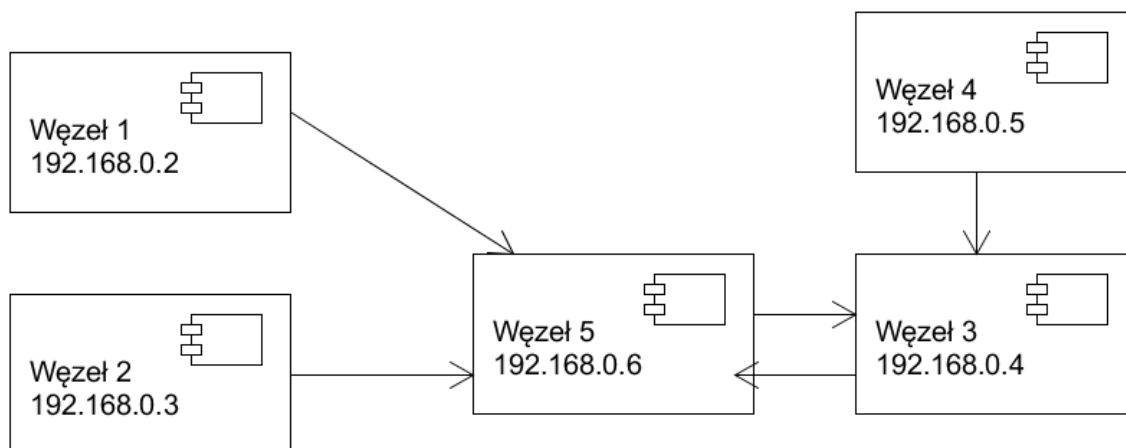
Na tym etapie projektowanie węzła jest zakończone i można przystąpić do instalacji oraz konfiguracji oprogramowania bazodanowego.

5.3. Instalacja i konfiguracja węzła

Program wykonywalny bazy danych dostępny jest w postaci pliku *jar* zawierającego zastosowane zewnętrzne biblioteki³⁰. Wymagane jest skopiowa-

²⁹czyli liczby całkowitej od -9223372036854775808 do 9223372036854775807 włącznie

³⁰Apache Commons IO oraz Google GSON, dzięki czemu nie jest wymagana ich osobna instalacja.



Rysunek 22: Przykład układu węzłów startowych. Strzałka oznacza węzeł, który dany węzeł ma za startowy (np. węzeł 1 łączy się do węzła 5).

nie tego pliku, wraz z referencyjnym plikiem konfiguracyjnym *config.json* do jednego katalogu. Katalog ten będzie katalogiem uruchomieniowym dla procesu bazy danych.

Po skopiowaniu plików do katalogu uruchomieniowego należy otworzyć w edytorze tekstowym plik *config.json*. Zawiera on szereg pól, które objaśniono w tabeli 6.

Rozbicie danych na trzy grupy: dane zasadnicze, metadane i ciągi pomocnicze, pozwala dostosować system plików pod konkretny typ danych, rozpatrując to jak często będzie do nich realizowany dostęp. Do zasadniczych danych zwykle będzie to zapis, a starsze pliki będą odczytywane stosunkowo rzadko. Dobrym rozwiązaniem może być dla tego typu danych system plików z kompresją. Z kolei metadane są małymi plikami, które muszą być załadowane względnie szybko, więc można im przydzielić szybki system plików oparty na urządzeniach SSD. W przypadku instalacji na systemie Windows należy pamiętać o stosowaniu znaków escape (np. *C : \\\ściezka* zamiast *C : \ściezka*). Należy upewnić się, że katalogi podane w konfiguracji rzeczywiście istnieją, a baza będzie miała prawo do ich zapisu.

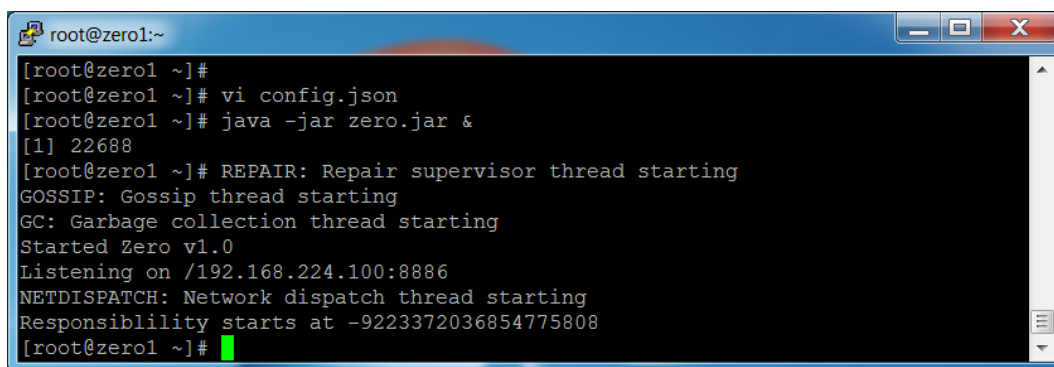
Pole *series_in_memory* należy dostosować do rozmiaru pamięci RAM w systemie, a przede wszystkim do liczby maksymalnych otwartych de-

Tabela 6: Pola konfiguracyjne pliku *config.json*

Nazwa	Opis
bootstrap_node_ip	Adres IP lub nazwa hosta węzła startowego.
bootstrap_node_port	Port, na którym nasłuchuje węzeł startowy.
node_ip	Adres IP interfejsu sieciowego, na którym ma nasłuchiwać ten węzeł.
node_port	Port TCP, na którym ma nasłuchiwać ten węzeł.
nodehash	Początek przedziału tego węzła.
seriesdata_path	Ścieżka do katalogu, w którym przechowywane będą ciągi główne.
seriesmeta_path	Ścieżka do katalogu, gdzie przechowywane będą metadane.
seriesdata_repair_path	Ścieżka do katalogu, gdzie przechowywane będą ciągi pomocnicze, służące do naprawy.
gc_grace_period	Czas (w sekundach) bezpieczny służący do usuwania ciągów. Interpretację szerzej opisano w p. 4.2. Rozsądnie jest pozostawić domyślną wartość wynoszącą 1 tydzień. Wartość tego parametru powinna być identyczna na wszystkich węzłach danego klastra.
series_in_memory	Maksymalny rozmiar cache dla ciągów. Tyle metadanych ciągów maksymalnie będzie przechowywanych w pamięci operacyjnej.

skryptorów plików. Nie stanowi to dużego problemu w systemach Windows (brak twardych limitów deskryptorów [13]), jednak może stanowić kłopot w systemach uniksowych, w których takie limity są określone [3], acz mogą być modyfikowane. Wartość docelową należy ustalić biorąc 120% wartości *series_in_memory*.

Żeby uruchomić węzeł, należy przejść do katalogu uruchomieniowego i wykorzystać do uruchomienia maszynę Javy. Przykład uruchomienia węzła będącego startowym przedstawiono na rysunku 23.



```
root@zero1:~#
[root@zero1 ~]# vi config.json
[root@zero1 ~]# java -jar zero.jar &
[1] 22688
[root@zero1 ~]# REPAIR: Repair supervisor thread starting
GOSSIP: Gossip thread starting
GC: Garbage collection thread starting
Started Zero v1.0
Listening on /192.168.224.100:8886
NETDISPATCH: Network dispatch thread starting
Responsibility starts at -9223372036854775808
[root@zero1 ~]#
```

Rysunek 23: Uruchomienie węzła startowego „zero1“

Celem wyjaśnienia sposobu projektowania i instalacji systemu przedstawiono tutaj konfigurację systemu, który posłużył później do celów testowych. Składać będzie się on z trzech węzłów o identycznych parametrach sprzętowych — jednordzeniowy procesor Intel³¹ o taktowaniu 2.3 GHz, dysk twardy 15 GB oraz 1 GB pamięci RAM. Systemem operacyjnym będzie GNU/Linux CentOS 7. Na dane przeznaczymy ok. 10 GB, biorąc pod uwagę konieczność pozostawienia miejsca dla systemu operacyjnego.

Pierwszym etapem projektowania jest stworzenie tabeli na wzór 5. Biorąc pod uwagę to że wszystkie węzły są równorzędne, przestrzeń wartości (równą 2^{64}) można rozdzielić między nie równo. Wyniki tego etapu zamieszczono w tabeli 7.

Kolejnym krokiem jest ustalenie grafu węzłów startowych. Ze względu na prostotę tej instalacji można określić jeden węzeł startowy — niech

³¹Maszyny zostały zrealizowane jako maszyny wirtualne pracujące w środowisku QEMU/KVM. Podczas testów przydzielono maszynom maksymalny priorytet na poziomie hipernadzorcy celem minimalizacji zakłóceń.

Tabela 7: Projekt klastra testowego (przykładowego)

IP	Miejsce na dane	Wartość przedziału
192.168.224.100	10 GB	-9223372036854775808
192.168.224.101	10 GB	-3074457345618258604
192.168.224.102	10 GB	3074457345618258601

będzie nim 192.168.224.100. Pozostałe węzły będą się łączyć do niego celem początkowej „koordynacji”.

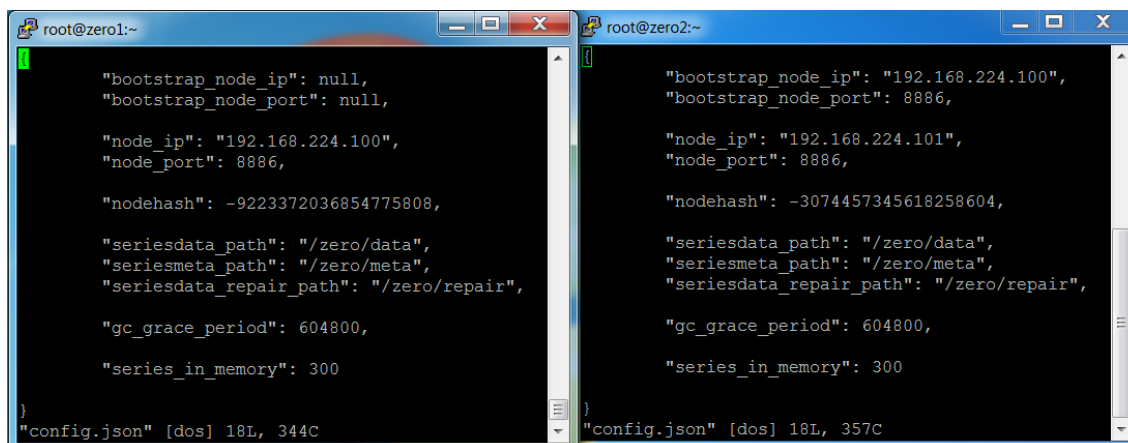
Po instalacji docelowego oprogramowania na maszynach przeprowadzono ich konfigurację. Była ona na wszystkich węzłach podobna, wystąpiły jedynie różnice w adresacji IP interfejsów oraz początków przedziałów. Drugą istotną różnicą było uzupełnienie ustawień węzła startowego w 192.168.224.101 i 192.168.224.102. Konfigurację przedstawiono na rysunku 24. Konfigurację 192.168.224.102 utworzono analogicznie.

Teraz należy przeprowadzić uruchomienie węzłów. Jako pierwszy powinien zostać uruchomiony węzeł 192.168.224.100. Jest to wymagane, gdyż węzły które będą uruchamiane później muszą być się w stanie z nim połączyć³². Tak więc węzły uruchomiono w kolejności 100, 101, 102. Wraz z każdym uruchomieniem możemy obserwować jak wszystkie węzły aktualizują swoją bazę danych, wykrywając nowe węzły. Większą pauzę (kilka sekund) zaobserwować można między startem 102 a wykryciem go przez 101, co związane jest z tym że propagacja przez węzeł 100 może zająć do 4 sekund. Stan konsoli po uruchomieniu węzła 101 w momencie gdy informacje są już rozesłane przedstawia rysunek 25.

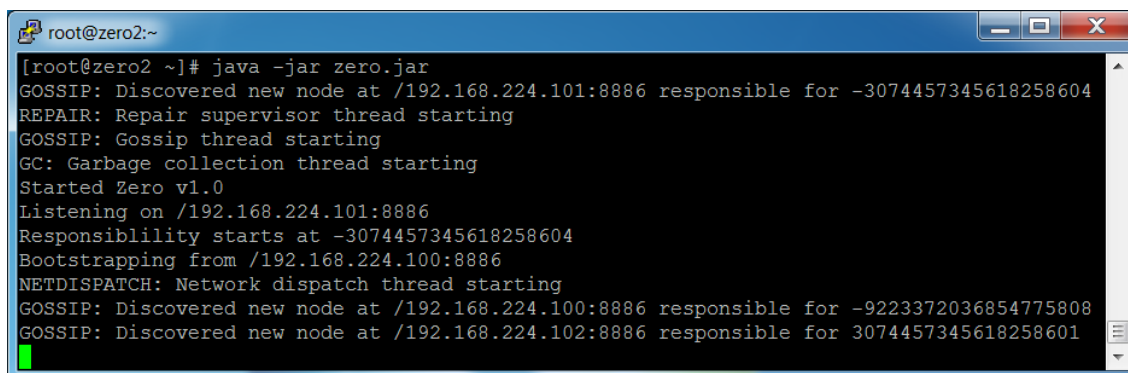
W tym momencie — czyli gdy każdy węzeł wie o każdym — klaster gotów jest do pracy w roli bazy danych systemu kontrolno-pomiarowego, a węzły oczekują na komendy.

Zatrzymania programu dokonuje się poprzez wymuszone „zabicie” procesu. Program jest zaimplementowany jako *crash-only software* [1], to jest oprogramowanie którego jedynym sposobem zatrzymania jest nagłe i

³²celem uproszczenia wywodu wcześniej poprawnie skonfigurowano firewall. Opis tej operacji wykracza poza temat niniejszej pracy.



Rysunek 24: Konfiguracja dwóch węzłów. Po lewej stronie konfiguracja 192.168.224.100, po prawej 192.168.224.101.



Rysunek 25: Konsola węzła 192.168.224.101 po uruchomieniu węzła 192.168.224.102. Węzeł najpierw „odkrywa” siebie samego (101), później odkrywa węzeł startowy (100) gdy łączy się do niego, a na samym końcu otrzymuje informację o węźle 102, który dołączył do klastra jako ostatni.

wymuszone przerwanie pracy. Przerwanie pracy programowi w dowolnej chwili, czy to przez awarię czy przez zatrzymanie z woli użytkownika, nie grozi nieodwracalnym uszkodzeniem danych.

Skonfigurowany tutaj przykładowy klastr posłuży później do testów.

5.4. Naprawa klastra i węzła

W pracy klastra może wydarzyć się sytuacja, że jeden z węzłów ulegnie awarii i przestanie reagować na komendy. Może być to związane z awarią sprzętową lub programową na systemie działającym jako węzeł.

Niezależnie od przyczyny awarii, obsługa bazy w przypadku awarii

jest bardzo prosta. Sama usterka może być dłuższy czas niezauważona, ze względu na wbudowaną tolerancję na awarie węzłów klastra. Z tego powodu zalecane jest korzystanie z zewnętrznych narzędzi monitorujących pracę poszczególnych węzłów. Są one standardowymi elementami wdrożenia systemów klastrowych.

Pamiętać należy o tym, że jedynym elementem identyfikującym węzeł w systemie jest początek jego przedziału odpowiedzialności. Inne węzły potraktują nawet nowy komputer, legitymujący się tą samą wartością przedziału, jako stary element klastra.

Gdy awaria została już usunięta na poziomie sprzętowo-systemowym, tj. jednym elementem który należy przywrócić jest działanie procesu bazy danych, należy rozpatrzyć czas awarii. Inny jest sposób postępowania w przypadku, gdy sumaryczny czas awarii węzła był mniejszy od czasu bezpiecznego (związanego z kasowaniem metadanych). Jeśli czas awarii był mniejszy, to należy po prostu uruchomić proces węzła i pozwolić mu samemu zsynchronizować się z resztą klastra. Należy jedynie upewnić się że w pliku konfiguracyjnym znajduje się taka sama wartość przedziału jak w węźle który uległ awarii.

Jeśli natomiast czas awarii był większy, lub system uległ awarii do tego stopnia że niezbędne było przywrócenie systemu z kopii zapasowej czy nowa instalacja, należy skonfigurować bazę „na czysto“, pamiętając jedynie o zgodności wartości przedziału z węzłem który uległ awarii. Jeśli system ocalał, a jedynie czas awarii był większy od czasu bezpiecznego, wystarczy wyczyścić katalogi danych, metadanych i ciągów naprawczych, a następnie uruchomić program.

5.5. Projektowanie i zapis ciągów

W klasycznych bazach danych SQL projektowanie tzw. schematu bazy danych jest zagadnieniem złożonym. W prezentowanej bazie, ze względu na jej relatywną prostotę, jest to wyzwanie nieco prostsze, jednak w dalszym ciągu nietrywialne. Przedstawiono tutaj kilka zasad i propozycji dotyczących się definiowania i planowania ciągów.

Przede wszystkim pierwszym zadaniem powinna być analiza systemu pomiarowego. Jeden ciąg bazy odpowiada najczęściej jednemu punktowi pomiarowemu, zbieranemu przez jeden czujnik. Błędem byłoby stworzenie jednego ciągu dla dwóch czy trzech czujników, zwłaszcza gdy ich działanie nie jest za sobą zsynchronizowane. Dopuszczalna jest natomiast sytuacja czy jeden ciąg reprezentuje dwa pomiary, które ze względu na konstrukcję systemu muszą być wykonywane jednocześnie i obsługiwane przez jeden komponent.

W ogólności, w systemie musi być określona odpowiedzialność elementu zapisującego za konkretny ciąg. Nie może występować sytuacja w której dwa elementy próbują zapisać do jednego ciągu. Związane jest to z tym, że element zapisujący musi pamiętać ostatni stempel czasowy zapisu, natomiast przy więcej niż jednym elemencie taka synchronizacja jest niemożliwa lub bardzo trudna. Nie stanowi ono problemu w systemie SMOK, gdyż agent obsługi obiektu odpowiedzialny jest za wszystkie ciągi danej lokalizacji. Jest to jednocześnie jedyny element który będzie zapisywał dane na rzecz tego obiektu i jest uruchomiony ciągle, doskonale spełnia więc te założenia.

Praktycznym przykładem planowania schematu bazy może być prosty system grzejąco-chłodzący składający się z grzejnika, klimatyzatora oraz czujnika temperatury wewnętrznej. Wartości z czujnika temperatury dopisywane byłyby do jednego z ciągów. Odnośnie stanów binarnych grzejnika i klimatyzatora (włączony lub wyłączony) możliwe są dwa rozwiązania. W pierwszym z nich stany każdego z urządzeń tworzyłyby osobne ciągi, w drugim oba stany zapisane na kolejnych bitach liczby dopisywane byłyby do wspólnego ciągu.

Baza danych posługuje się dedykowanym protokołem dostępu, niezbędne jest więc zastosowanie elementu „zbieracza” danych. Jego zadaniem byłoby przetwarzanie protokołu czujników i systemu automatyki na protokół bazy. Pełniłby on także rolę „bufora” danych, zapewniając że żadne dane nie zostaną utracone podczas zapisu, oraz pełniłby rolę abstrakcji dla nietypowego sposobu zapisu wykorzystywanego przez bazę

(konieczność pamiętania definicji ciągu i ostatniego stempla czasowego zapisu). Rolę takiego elementu w systemie SMOK pełni agent obsługi obiektu. Działając cały czas jest w stanie buforować zapisy.

Innym zagadnieniem może być „symulacja” rekordów o różnych długościach. Ponieważ ciąg pozwala jedynie na rekordy o stałej długości, można go zasymulować definiując dwa ciągi. Załóżmy że mamy system rejestrujący wartości zespolone lub wielowymiarowe (np. przesuw frezarki, odczyty zespołu akcelerometrów), przy czym znakomita większość pomiarów to wartości rzeczywiste. Zamiast definiować pojedynczy ciąg zawierający wartości zespolone o dwukrotnie większym rozmiarze, można zdefiniować dwa ciągi: jeden od części rzeczywistych, drugi od urojonych. Ponieważ większość pomiarów będzie rzeczywista, to będą one trafiać jedynie do ciągu rzeczywistego. Jeśli pojawi się jakaś część urojona, może być ona dopisana do ciągu liczb urojonych, z identycznym stemplem czasowym co odpowiadająca jej część rzeczywista. Ideę takiego rozwiązania przedstawiono na rysunku 26.

<i>stempel</i>	1	2	3	4	5	6	7	8	9
<i>wartość</i>	45	23	18	11	75	45	32	16	9

<i>stempel</i>	1	2		5		8
<i>wartość</i>	j23	j11		j18		j2

Rysunek 26: Przykład implementacji „rekordu o zmiennej długości”. Pomiar dla stempla czasowego 1 można przedstawić jako $45 + j23$, zaś dla stempla 3 jako 18.

Określenie precyzyjnych schematów i wytycznych nie jest możliwe do określenia ścisłymi schematami i wytycznymi, podobnie jak dla baz relacyjnych i wymaga w dużej mierze doświadczenia i intuicji.

```

from zerocon import ZeroInterface, ZeroException, \
    SeriesNotFoundException, \
    DefinitionMismatchException

zero = ZeroInterface([( '192.168.224.100', 8886),
                      ( '192.168.224.101', 8886),
                      ( '192.168.224.102', 8886)])

try:
    sd = zero.getDefinition('test.series')
    head = zero.getHeadTimestamp(sd)

    zero.write(sd, head, head+1, '0000')
    zero.close()
except SeriesNotDefinedException:
    print 'Nie zdefiniowano takiego ciagu'
except ZeroException:
    print 'Wystapil inny problem'

```

Listing 6: Skrypt dopisujący rekord do ciągu

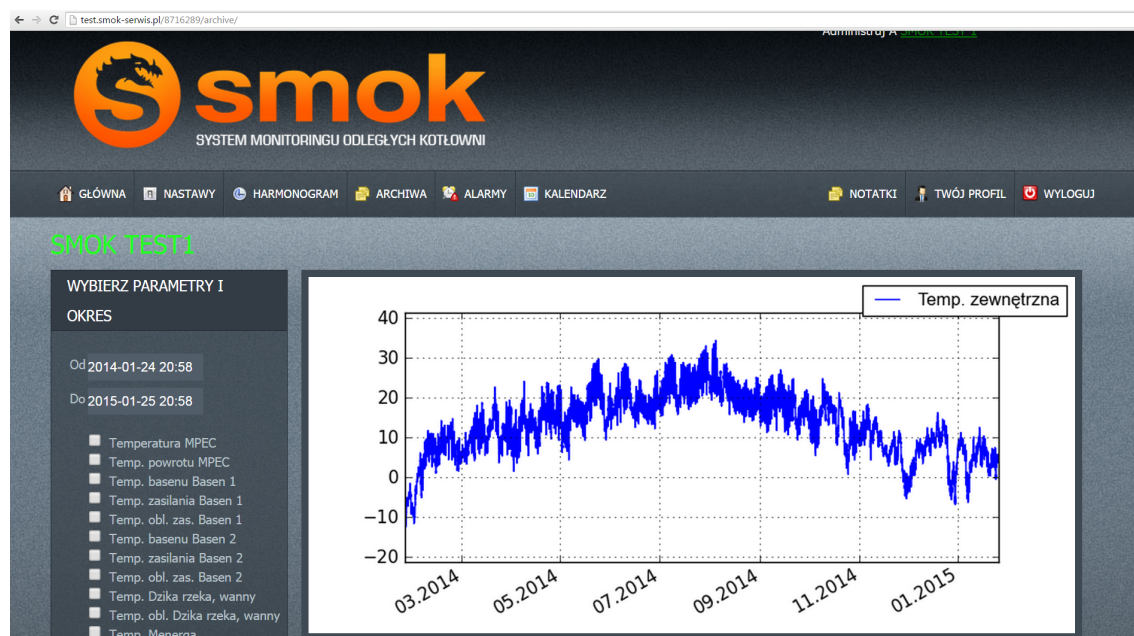
5.6. Interfejs Python

Celem demonstracji możliwości skorzystania z funkcji przygotowywanej bazy przy użyciu języków programowania innych niż Java, stworzono także interfejs napisany w języku Python. Sprawdza się on jako język służący do zespalandia różnych elementów systemu. Charakteryzuje się ponadto szybkością tworzenia oprogramowania. Również, idąc z duchem języka, stworzony interfejs charakteryzuje się prostotą. Na listingu 6 przedstawiono program dopisujący wartość do ciągu. Oczywiście nie jest konieczne wypisywanie wszystkich węzłów, ponieważ wystarczy jedynie kilka aby interfejs skorzystał z innego, jeśli wystąpią problemy z połączeniem.

Interfejs dysponuje również opcją wyrównywania obciążeń — łączy się najpierw do losowego hosta z podanej listy, ze względu na ekonomię skali powinno wyrównywać obciążenie węzłów.

Znaczące części systemu SMOK stworzone zostały właśnie w języku Python. Tyczy się to zwłaszcza komponentów które odpowiedzialne są za zapis pomiarów w obecnej bazie danych. Dzięki temu integracja tworzonego rozwiązania z resztą systemu jest prostsza niż w przypadku skorzystania z interfejsu dostępnego w innym języku programowania. Wymagałoby to

dodatkowych nakładów pracy. Przeprowadzono integrację bazy danych z testową wersją systemu SMOK, umieszczając w niej wcześniej zebrane dane. Na rysunkach 27 i 28 przedstawiono zrzuty ekranu z pracy testowej bazy w systemie SMOK.



Rysunek 27: Zrzut ekranu z pracy bazy na testowej platformie SMOK. Sporządzono wykres temperatury zewnętrznej od końca stycznia 2014 do 2015.



Rysunek 28: Zrzut ekranu z pracy bazy na testowej platformie SMOK. Sporządzono wykres temperatur obiegów centralnego ogrzewania.

6. Testy systemu

Zdecydowano się na przeprowadzenie trzech testów, które miały sprawdzić różne aspekty pracy stworzonego systemu bazodanowego. Pierwszym z nich był test określający w jaki sposób system reaguje na wzrost obciążenia. Wyniki testu pozwolił ustalić komponent systemu, który ma największy wpływ na szybkość rozwiązania. Drugi test pozwalał na określenie zachowania systemu w trakcie awarii, oraz jego możliwości samonaprawy. Trzeci test miał na celu stwierdzenie czy rozwiązanie nadaje się wydajnościowo do zastosowania w systemie kontrolno-pomiarowym SMOK.

Środowisko testowe jest identyczne z tym, którego konfigurację przedstawiono w rozdziale 5.3. Należy zauważyć, że skonfigurowane elementy są maszynami wirtualnymi, tak więc w celu minimalizacji zakłóceń pomiarowych związanych z obciążeniem fizycznej maszyny ustalono im priorytet wyższy niż pozostałym maszynom wirtualnym na tym serwerze.

Dodatkowo, skonfigurowano wirtualną maszynę kliencką o adresie 192.168.224.103. Będzie ona kierować zapytania do klastra przy użyciu interfejsu Python.

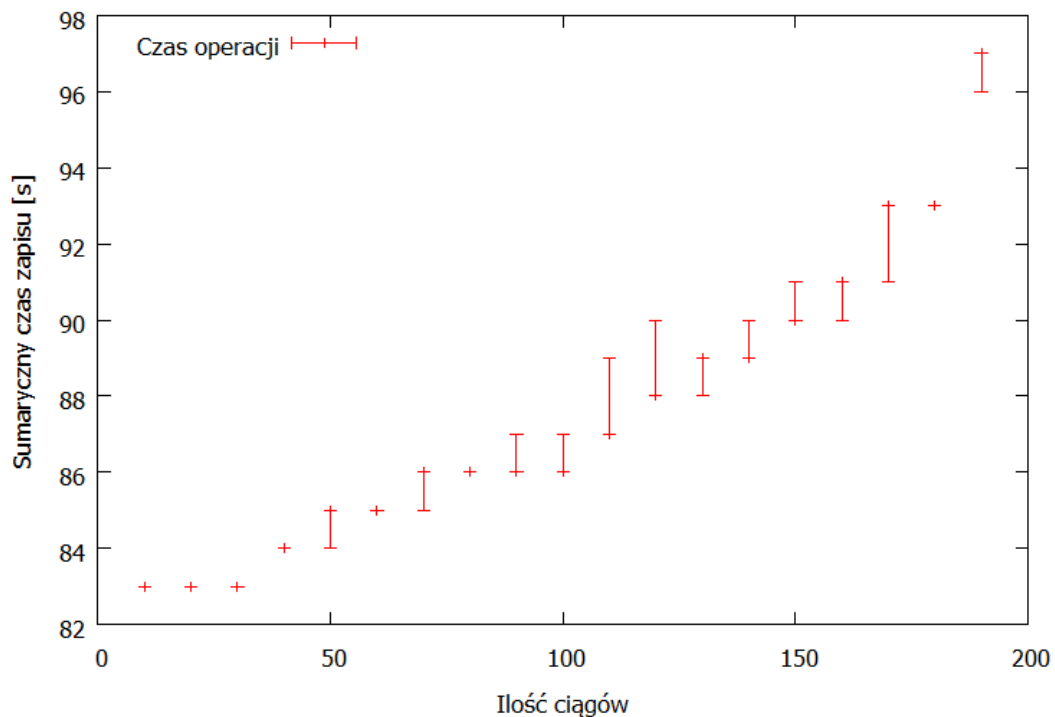
6.1. Zależność czasu obliczeń od liczby ciągów

Pierwszym testem, który przeprowadzono, był test skalowalności mający na celu zbadanie właściwości zbudowanej bazy pod względem skalowalności. Testem tym był równoległy zapis do wielu ciągów. Pomiar dla pojedynczej ilości ciągów polegał na stworzeniu pewnej liczby (parametryzowanej) wątków systemu operacyjnego. Zadaniem każdego z wątków było zapisanie 1000 rekordów do własnego ciągu, to jest każdemu wątkowi przyporządkowany był jego własny ciąg i własne połączenie TCP. Następnie, gdy wątki zostały utworzone, zostały w jednej chwili uruchamiane, od tego momentu liczony był sumaryczny czas zapisu. W momencie gdy ostatni wątek zapisujący kończył, czas był kończony.

Każdy ciąg miał 2 repliki i rozmiar pola danych 4 bajty. Pomiaru dla każdej ilości ciągów dokonywano 3 razy, wybierając medianę. Badano tutaj

zależność czasu wykonania całej operacji od ilości równoczesnych zapisów. Oczekuje się, że wąskim gardłem systemu będą operacje I/O, tak więc fakt że węzły mają po jednym procesorze nie będzie problemem.

Zapis realizowano poprzez klienta wielowątkowego napisanego w języku Python.



Rysunek 29: Wyniki pomiaru zależności ilości równoległych zapisywanych ciągów od czasu (w sekundach) trwania całej operacji. Słupki błędów oznaczają minimum i maksimum.

Wyniki eksperymentu przedstawiono w postaci wykresu na rysunku 29. Widoczna jest korelacja dodatnia ilości ciągów i czasu całej operacji. Sumarycznie jest ona jednak bardzo niewielka, bo stanowi przyrost o 16%³³ przy 19-krotnym zwiększeniu obciążenia. Test został zaprojektowany w celu ustalenia charakteru zrównoleglenia stworzonej bazy danych. Cele te zostały osiągnięte, wyniki bowiem należy zinterpretować następująco:

1. Rozwiązanie jest bardzo dobrze zrównoleglone pod względem obliczeniowym

³³ $\frac{97,10s - 83,10s}{83,10s}$

2. Prędkość rozwiązania zależy w głównej mierze od przepustowości sieciowej

Punkt 1 oznacza, że baza danych nie generuje znaczącego obciążenia w obszarze obliczeń i dostępu do dysku. Przyrost czasu na obliczenia był bowiem niewielki mimo znaczącego wzrostu obciążenia. Oznacza to, że stopień optymalizacji kodu jest zadowalający. Punkt 2 wymaga szerszego omówienia. Węzły posiadały procesory jednordzeniowe, czyli system miał łącznie 3 procesory, a mimo tego nie zaobserwowano znaczącego przyrostu czasu. Większość czasu wątki spędzały w uśpieniu, oczekując albo na operację I/O dysku, albo na operację sieci. Spodziewano się przynajmniej liniowego przyrostu czasu. Równoległość w tym teście polegała na zrównoleglaniu przez system operacyjny operacji dyskowych i sieciowych (system nie musi wykonywać ich w złeconej kolejności, ale w kolejności optymalnej). Ilość zapisanych bajtów na dysk była niewielka³⁴, a na jeden ciąg wykonano tylko jedną operację otwarcia pliku ze względu na obecność podsystemu cache. Pozwala to wykluczyć operacje dysku jako źródło wąskiego gardła. Dominującym czynnikiem musiały być tutaj operacje sieciowe, to jest transmisja danych między węzłami, oraz między węzłami a klientem. Wnioskiem testu jest więc to, że przyspieszenie pracy sieci przyspiesza działanie klastra szybciej niż ulepszenie procesora czy dysku. Rozmieszczenie klastrów na osobnych fizycznych maszynach z całą pewnością spowolni pracę systemu, jednak nie dysponowano infrastrukturą na której można by określić jak.

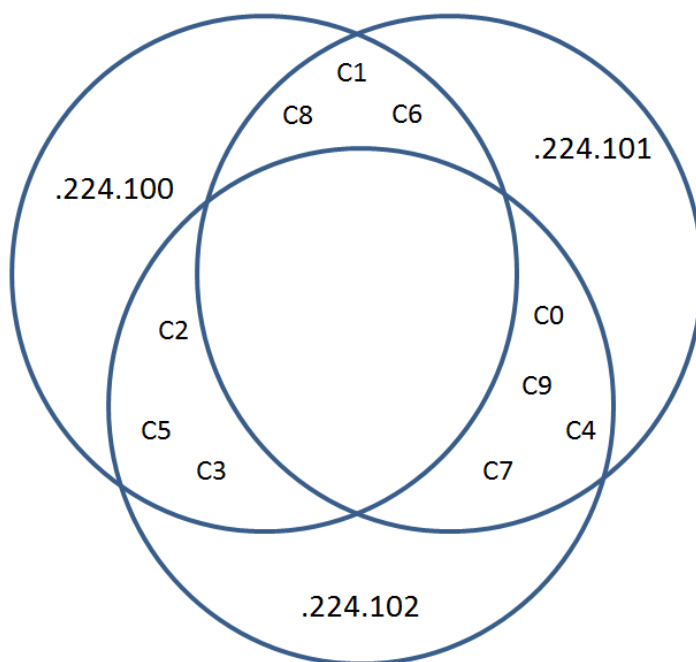
6.2. Reakcja systemu na awarię

Celem sprawdzenia reakcji systemu na awarię przeprowadzono test polegający na zapisie 10 ciągów pomiarowych (C0 – C9), z zapisem co sekundę. Poziom replikacji każdego ciągu wynosił 2, co oznacza że w systemie obecne będą 2 kopie danych ciągu. W pewnym momencie wyłączono jeden z węzłów systemu (węzeł 3 o adresie 192.168.224.102), aby zasy-

³⁴ok. 2,2 MB

mulować utratę zasilania komputera³⁵. Sprawdzono wtedy czas między wysłaniem rozkazu zapisu a jego zrealizowaniem. Wszystkie zapytania testowy klient kierował do węzła .224.100.

Przeglądając katalogi z danymi ustalono które ciągi znalazły się na którym węźle. Przynależność węzłów do replik ujęto na diagramie Venna przedstawionym na rysunku 30.



Rysunek 30: Przynależność ciągów do węzłów

Dla każdego zapisu w każdym ciągu notowano trzy wartości: czas wygenerowania „pomiaru”, czas w którym zapis tego pomiaru zaczął być realizowany, oraz czas kiedy zapis został zakończony. Ponieważ awarii uległ węzeł 3 (.224.102), szczególnie interesujące będą wyniki dla węzłów przynależących do tegoż węzła. Ciągi należące tylko do węzłów 1 i 2 będą stanowić grupę kontrolną.

Test przebiegł zgodnie z procedurą:

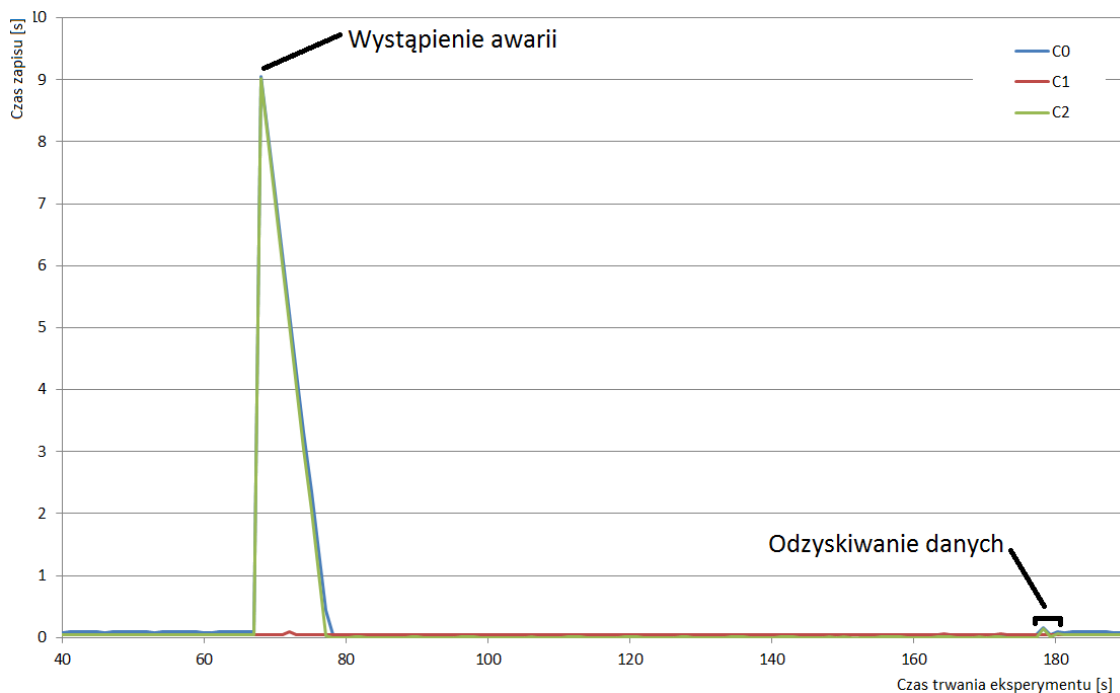
1. $T + 0\text{ s}$: Rozpoczęcie testu
2. $T + 68\text{ s}$: Awaria węzła 3
3. $T + 178\text{ s}$: Przywrócenie pracy węzła 3

³⁵przerwano pracę maszyny wirtualnej za pomocą polecenia `virsh destroy`.

4. $T + 220$ s: Koniec zapisów testowych

5. $T + 230$ s: Wyłączenie programów węzłów

Do zgrubnej analizy wybrano ciągi C0, C1 oraz C2 (każdy z innej grupy). Na podstawie zgromadzonych danych obliczono ile trwał jeden zapis. Poprzez „czas trwania zapisu” rozumiemy tu okres między pobraniem „pomiaru”, a zgłoszeniem przez bazę że został w sposób trwały³⁶ zapisany. Wyniki przedstawiono na rysunku 31.



Rysunek 31: Czas trwania zapisu w funkcji czasu eksperymentu wyrażonego w sekundach

Stwierdzić można że przy awarii węzła 3 (68 sekunda) zaobserwowano dla pierwszego zapisu po awarii opóźnienie rzędu 10 sekund. Był to czas, w którym pozostałe węzły stwierdziły, że trzeci uległ awarii i zaktualizowały swoje tabele, aby uniknąć wysyłania temu węzłowi rozkazów zapisu. Czas wykonania zapisu stopniowo zmniejszał się — wszak przez 10 sekund zgromadzono aż 10 zapisów, które musiały być nadgonione. Potem, aż do przywrócenia działania, czas zapisu nie różni się znacząco od stanu

³⁶Oznacza to, że jeśli po tym momencie wystąpi awaria, to dane przetrwają w bazie danych.

sprzed awarii. W momencie przywrócenia węzła (178 sekunda) obserwuje się krótki pik, związany z obciążeniem systemu próbującego przywrócić brakujące dane. W tym przypadku pobranie brakujących danych przez węzeł 3 z węzłów 1 i 2 zajęło dokładnie 5 sekund³⁷. W obu przypadkach awaria węzła 3 nie miała większego wpływu na opóźnienia w zapisie ciągu C1, który to był obsługiwany przez sprawne węzły 1 i 2. Opóźnienie C1 nigdy nie zwiększyło się powyżej 1, co oznacza że zapisy nie musiały być kolejkowane. Powolny opad piku tłumaczy to że czas liczony był od pobrania pomiaru, więc naturalne pojedyncze opóźnienie opóźni kilka następnych zapisów. Tylko pierwszy zapis po awarii jednak przetwarzany był ok. 10 sekund. Był to czas wymagany na stwierdzenie awarii węzła.

Dziesięciosekundowa przerwa z punktu widzenia systemu SMOK nie jest żadną przeszkodą. Agent obsługi obiektu jest w stanie buforować dane do zapisu przez ten czas, co nie wpłynie negatywnie na jakość świadczonych usług.

Przy końcu eksperymentu porównano dane zgromadzone na węzłach. Wszystkie repliki zgadzały się, co oznacza że system zadziałał w sposób poprawny, a baza jest w stanie naprawić awarię.

6.3. Test obciążeniowy

Najważniejszym testem, sprawdzającym zdolność bazy do pracy w systemie, był test obciążeniowy. W p. 3.3 ustalono maksymalną liczbę urządzeń i punktów pomiarowych obsługiwanych przez bazę. Niestety, nie dysponowano sprzętem, podobnym do tego na którym w przyszłości mógłby być uruchamiany system bazodanowy. Zdecydowano się więc na symulację wariantu systemu SMOK dla roku 2020 z prognozą dodawania urządzeń co 2 dni. Byłyby to więc 1432 urządzenia, każde po 14 punktów pomiarowych. Dodatkowo, maszyny wirtualne stanowiące składniki klastra wyposażono w drugi procesor wirtualny³⁸. Upewniono się ponadto, że wirtualne urzą-

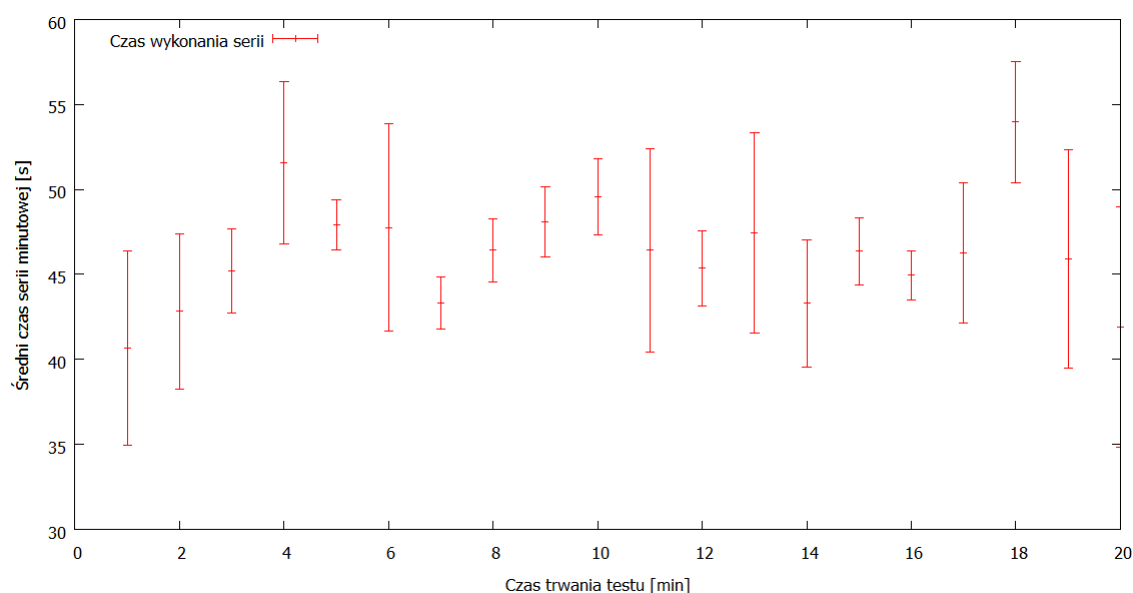
³⁷Nie jest to widoczne na wykresie zaprezentowanym na rysunku 31. Raport ten uzyskano na podstawie raportu bazy podczas przeprowadzania eksperymentu.

³⁸Upewniono się że sumaryczna liczba procesorów nie przekraczała liczby wątków procesora na systemie hipernadzorcy.

dzenia nie są parawirtualizowane, gdyż przekłamywałyoby to testy.

Słowem, stworzono testowego klienta uruchamiającego 1432 połączenia do bazy. Każde połączenie oznaczało zapis co minutę 14 pomiarów do bazy. Ten test miał zweryfikować, czy da się wykonać 1432 połączenia w ciągu minuty. W przypadku przekroczenia czasu zgłaszany był błąd. Niemożność wykonania 14 zapisów na połączenie w ciągu minuty oznaczałaby, że rozwiązanie nie jest w stanie wytrzymać obciążenia. Eksperyment trwał 20 minut. Zdecydowano bowiem (jest to pewne uproszczenie) że jeśli baza wytrzyma to obciążenie przez 20 minut, to będzie je w stanie wytrzymać dowolnie długo. Baza musiała więc być w stanie obsłużyć 20048 zapisów na minutę, to jest ok. 334 zapisów na sekundę.

Czas trwania każdej serii minutowej (to jest wykonania 14 zapisów w obrębie danego połączenia) był zapisywany do pliku. Otrzymano więc 1432 plików zawierających 20 pozycji. Raporty te obrobiono do postaci prezentującej maksimum, minimum oraz wartość średnią. Zaprezentowano je w formie wykresu na rysunku 32.



Rysunek 32: Wykres obrazujący wyniki testu obciążeniowego

W jednym tylko przypadku, to jest 18 minuty, czas maksymalny zbliżył się do twardego limitu 60 sekund. W pozostałych przypadkach wartości były poniżej tego progu. Oznacza to, że rozwiązanie to było w stanie podołać założeniom testu. Powziąć można hipotezę, że gdyby dysponowano

sprzętem o większej wydajności, możnaby przeprowadzić test z jeszcze większą liczbą urządzeń. Wyniki wskazują również na pewien zapas w czasie, który baza mogłaby spożytkować na „nadgonienie” zapisu danych w przypadku awarii klastra. Baza bowiem sama z siebie wygenerowałaby opóźnienie rzędu ok. 10 sekund, potrzebnych na identyfikację węzła bazy który uległ awarii.

Pozytywny wynik testu potwierdza możliwość zastosowania rozwiązania w systemie kontrolno-pomiarowym SMOK.

7. Podsumowanie

Problem przechowywania danych w postaci szeregów czasowych jest problemem istotnym. W dobie informacji dokładne opomiarowanie procesu technologicznego, a także przechowywanie i analiza tych danych stanowić może o silnej przewadze nad konkurencją. Wolumen zbieranych danych zwiększa się coraz bardziej, dlatego stabilne i wydajne rozwiązanie bazodanowe stanowi niezbędne uzupełnienie nowoczesnego systemu automatyki przemysłowej.

Rozwiązania typowo przemysłowe, korzystające ze standaryzowanych interfejsów, coraz częściej wewnętrznie przypominają rozwiązania oparte o systemy Big Data. Działania takie są niezbędne by radzić sobie wydajnie z ilością danych, której tradycyjne systemy nie są w stanie przetworzyć. Z drugiej strony, nietrudno wyobrazić sobie adapter OPC do rozwiązania Big Data, pozwalający z bazy, której przeznaczeniem był nadzór aplikacji internetowej, zbudować system kontrolno-pomiarowy. Jest to możliwe, zwłaszcza że w obu przypadkach bazy przetwarzają szeregi czasowe. W pewnych warunkach rozwiązania te są tożsame.

Zaprezentowane w pracy rozwiązanie, w kontekście sprzęgnięcia z systemem SMOK, charakteryzuje szereg zalet w stosunku do istniejących rozwiązań tego typu. Są to:

- prosta konfiguracja i instalacja bazy,
- redundancja bez konieczności stosowania systemów RAID,
- niewielkie wymagania programowe, konieczna jedynie jest maszyna wirtualna języka Java.
- dostępność kodu źródłowego i możliwość rozwijania oprogramowania bez ponoszenia kosztów opłat licencyjnych,
- łatwa integracja z systemem SMOK (interfejs w języku Python).

W pracy przedstawiono rozwiązanie problemu zawierające nowe jakościowo elementy. Zastosowanie algorytmu tabeli mieszającej do kojarzenia

węzłów z ciągami spowodowało, że dla użytkownika najtrudniejszą częścią zaprojektowania klastra jest wykonanie kilku prostych działań matematycznych, a konfiguracji — skopiowanie dwóch plików i dokonanie niewielkich korekt w jednym z nich. Jedynym wymaganiem programowym jest maszyna wirtualna Java, dzięki czemu rozwiązanie działa równie dobrze na systemach Windows co zgodnych z POSIX.

Rozwiązanie to jest autorskie, co umożliwia twórcy pełną kontrolę nad jego rozwojem. Będzie on niezbędny w miarę rozwoju systemu SMOK. Dzięki zastosowaniu bardzo prostej metody zapisu (pliki są ciągami stempli czasowych i wartości pomiarów) można przetwarzać te dane nawet bez konieczności bezpośredniego odpytywania węzła — wystarczy dostęp do pliku.

Nowatorska metoda sprawdzania spójności ciągu (obowiązek pamiętania definicji szeregu oraz ostatniego stempla czasowego) spowodowała istotne uproszczenie systemu, bardzo szybką detekcję problemów. Pozwala ona przywrócić węzeł, wraz z kompletem danych, po awarii w czasie rzędu kilkunastu minut, nawet w przypadku fizycznego zniszczenia węzła. Pozwala to także na nieprzerwaną pracę w sytuacji, kiedy część danych w klastrze jest fizycznie niedostępna (np. rozległa awaria sieci). Dane te baza przywróci, gdy awaria zostanie usunięta.

Różnicą w stosunku do rozwiązań opartych o Big Data i klasyfikujących tworzoną bazę jako rozwiązanie dla automatyki jest silna gwarancja przetrwania zapisywanych do niej danych, mimo potencjalnej awarii. Opracowana baza zwraca bowiem potwierdzenie zapisu dopiero, gdy dane zostaną zapisane na nośnik. Dodatkowo, zintegrowana funkcja kasowania starych danych pozwala administratorom na uproszczone przewidywanie ilości miejsca dyskowego, które maksymalnie zużyje klaster. Zapewnia to także możliwość zastosowania rozwiązania do monitorowania aplikacji sieciowych, jak i do przechowywania pomiarów systemu kontrolno-pomiarowego przez zadany okres.

Przeprowadzono również testy. Pierwszy z nich miał na celu określenie charakteru obciążenia, które generuje rozwiązanie. Stwierdzono że

największym ograniczeniem dla szybkości działania stworzonej bazy są opóźnienia generowane przez transmisję sieciową między węzłami. Przeprowadzono również test ustalający zachowanie się bazy podczas awarii. Stwierdzono, że zdarzenie takie wygeneruje opóźnienie ok. 10 sekund. Czas ten jest potrzebny na stwierdzenie, że węzeł uległ awarii i nie odpowiada na polecenia. Ostatnim testem był test obciążeniowy, którego wyniki potwierdzają że opracowane rozwiązanie może być z powodzeniem zastosowane w systemie SMOK, nawet w przypadku opóźnienia wygenerowanego przez awarię węzła.

Pewnym mankamentem testów była konieczność przeprowadzenia ich na maszynach wirtualnych, ze względu na niedostępność odpowiedniej klasy sprzętu. Testowe maszyny wirtualne miały dużo mniejszą wydajność od rzeczywistych systemów, na których baza byłaby uruchamiana. Zachowano jednocześnie, na platformie testowej, opóźnienia sieciowe charakterystyczne dla oddalonych systemów. Wpływa to przekłamująco na wyniki, nie można jednak jednoznacznie stwierdzić czy na korzyść czy niekorzyść rozwiązania.

Rozwiązanie, aby stać się pełnowartościowym rozwiązaniem typu historian wymaga integracji ze standardami przemysłowymi. Niezbędne byłoby stworzenie mostów umożliwiających przekaz danych z sensorów i sterowników przemysłowych do bazy danych. Także bez tego, choć po dostosowaniu infrastruktury akwizycji, stworzona baza może być stosowana w dowolnym systemie kontrolno-pomiarowym — w ten sposób rozwiązanie będzie zintegrowane z systemem SMOK. Baza oczekuje obecnie na wdrożenie w produkcyjnej wersji zaprezentowanego w pracy systemu kontrolno-pomiarowego. Przede wszystkim dlatego, że przeszła ona pozytywnie wstępne testy, cel pracy można uznać za osiągnięty.

Autor za wkład własny pracy uważa:

- przegląd literatury związanej z zagadnieniem oraz zapoznanie się z obecnymi rozwiązaniami bazodanowymi,
- przeprowadzenie analizy wymagań rozwiązania w kontekście potrzeb systemu SMOK,

- zaprojektowanie i wykonanie programu rozproszonej bazy danych oraz modułu klienckiego,
- zaprojektowanie protokołu sieciowego niezbędnego do realizacji celów projektu,
- zaprojektowanie i przeprowadzenie testów weryfikujących poprawność działania rozwiązania.

Spis rysunków

1. Architektura rozwiązania OpenTSDB	13
2. Ilustracja metody drzew Merkle'a	19
3. Schemat blokowy nowej metody sprawdzania spójności . . .	20
4. Schemat ideowy przykładowego podziału przestrzeni wartości funkcji mieszającej na trzy węzły	21
5. Schematyczna architektura systemu SMOK	23
6. Przykładowy wykres danych, które mogą być zbierane przez stworzoną bazę	25
7. Schemat komunikacji w miejscu instalacji modułu	26
8. Schemat przepływu danych wewnątrz części serwerowej . . .	27
9. Zrzut ekranu interfejsu WWW z produkcyjnej wersji systemu SMOK	29
10. Schemat sekwencji odczytu pojedynczego parametru	30
11. Prognoza ilości danych zgromadzonych w systemie na lata 2015, 2016	33
12. Przykład ciągu o 5 rekordach z rosnącym stemplem czasowym	34
13. Diagram struktury przykładowej instalacji z trzema węzłami	36
14. Diagram ogólnej idei przepływu danych	37
15. Prezentacja modułów i ich wzajemnych powiązań	38
16. Graficzna interpretacja przydziału węzłów odpowiedzialnych za repliki, w zbiorze wartości funkcji mieszającej	42
17. Algorytm porównania i synchronizacji definicji	43
18. Algorytm detekcji niespójności ciągu	43
19. Ilustracja rozsyłania danych o węzłach za pomocą <i>gossip</i> . .	50
20. Graficzna reprezentacja możliwego stanu ciągu po dwóch awariach	61
21. Diagram UML interfejsu wraz z trzema istniejącymi klasami implementującymi go.	63

22. Przykład układu węzłów startowych. Strzałka oznacza węzeł, który dany węzeł ma za startowy (np. węzeł 1 łączy się do węzła 5).	74
23. Uruchomienie węzła startowego „zero1“	76
24. Przykład konfiguracji dwóch węzłów	78
25. Zrzut ekranu konsoli węzła	78
26. Przykład implementacji „rekordu o zmiennej długości“	81
27. Zrzut ekranu z pracy bazy na testowej platformie SMOK . .	83
28. Zrzut ekranu z pracy bazy na testowej platformie SMOK . .	83
29. Wyniki pomiaru zależności ilości równoległych zapisywanych ciągów od czasu trwania całej operacji	85
30. Przynależność ciągów do węzłów	87
31. Czas trwania zapisu w funkcji czasu eksperymentu wyrażonego w sekundach	88
32. Wykres obrazujący wyniki testu obciążeniowego	90

Spis tabel

1. Minimalna ilość miejsca na niekompresowanych danych przechowywanych przez 2 lata przy przykładowych rozmiarach systemów kontrolno-pomiarowych.	9
2. Zestawienie wspomnianych rozwiązań opartych na Big Data	13
3. Opis typów danych używanych w specyfikacji (typ zapisu <i>big endian</i>)	68
4. Opis definicji ciągu w reprezentacji sieciowej (pola podane w kolejności)	68
5. Przykład projektu małego klastra	73
6. Pola konfiguracyjne pliku <i>config.json</i>	75
7. Projekt klastra testowego (przykładowego)	77

Bibliografia

- [1] Candea, George i Armando Fox: *Crash-Only Software*. W *Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems*, strony 67–72, 2003.
- [2] DMS Serwis s.c.: *System Monitoringu Odległych Kotłowni*. <http://www.smok-serwis.pl/>.
- [3] Gafton, Cristian: *limits.conf(5) - Linux man page*. <http://linux.die.net/man/5/limits.conf>.
- [4] GE Intelligent Platforms: *Proficy Historian 5.5*. <http://www.ge-ip.com/download/proficy-historian-5-5/12501/2420/>.
- [5] George, Lars: *HBase: The Definitive Guide*. O'Reilly Media, Kwiecień 2001, ISBN 978-1-4493-9610-7.
- [6] Honeywell: *Uniformance PHD*. <https://www.honeywellprocess.com/library/marketing/notes/Uniformance-PHD-PIN.pdf>.
- [7] Lakshman, Avinash i Prashant Malik: *Cassandra: A Decentralized Structured Storage System*. SIGOPS Oper. Syst. Rev., 44(2):35–40, Kwiecień 2010, ISSN 0163-5980.
- [8] Lawson, Stephen: *GE thinks it's time to put industrial data in the cloud*. <http://www.pcworld.com/article/2042373/ge-thinks-its-time-to-put-industrial-data-in-the-cloud.html>.
- [9] National Instruments: *Logging Data with National Instruments Citadel*. <http://www.ni.com/white-paper/6579/en/>, Lipiec 2012.
- [10] OPC Foundation: *OPC Historical Data Access Specification*. Raport techniczny, Grudzień 2009.
- [11] Planet Cassandra: *Apache Cassandra NoSQL Performance Benchmarks*. <http://planetcassandra.org/nosql-performance-benchmarks/>.

- [12] Rockwood, Ben: *Getting Started with RRDtool*. <http://cuddletech.com/articles/rrd/rrdintro.pdf>, Kwiecień 2004.
- [13] Russinovich, Mark: *Pushing the Limits of Windows: Handles*. <http://blogs.technet.com/b/markrussinovich/archive/2009/09/29/3283844.aspx>.
- [14] Shvachko, Konstantin, Hairong Kuang, Sanjay Radia i Robert Chander: *The Hadoop Distributed File System*. W *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, strony 1–10, Washington, DC, USA, 2010. IEEE Computer Society, ISBN 978-1-4244-7152-2.
- [15] Stoica, Ion, Robert Morris, David Karger, M. Frans Kaashoek i Hari Balakrishnan: *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*. W *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, strony 149–160, New York, NY, USA, 2001. ACM, ISBN 1-58113-411-8.

Dodatek

Na załączonej płycie CD umieszczono:

- tekst pracy w formacie PDF oraz \LaTeX ,
- kod źródłowy pracy,
- pliki zawierające wyniki pomiarów.