Mainframe i z/OS, część 4 z 9: Jak się z tym pracuje: JCL, datasety i zielony ekran

Posłuchaj strony

Posłuchaj strony

Jak się z tym pracuje: JCL, datasety i zielony ekran

Tu wchodzimy w codzienność. Ostrzegam: to jest część, która najbardziej różni się od wszystkiego, co znasz.

JCL — język, który nie wybacza

Job Control Language to nie powłoka. To deklaratywny opis zadania wsadowego: co uruchomić, na jakich zbiorach, z jakimi zasobami. Dziedzictwo karty dziurkowanej jest widoczne gołym okiem: `//` w kolumnach 1–2 oznacza kartę JCL, treść mieści się w kolumnach 3–71, znak kontynuacji w 72, komentarz zaczyna się od `//*`.

Najprostszy przykład — alokacja zbioru programem, który dosłownie nic nie robi:

`IEFBR14` to legendarny program-atrapa: jego cały kod to instrukcja `BR 14`, czyli natychmiastowy powrót. Uruchamia się go wyłącznie po to, żeby system wykonał to, co opisują karty DD — alokację albo skasowanie zbioru. Program, który nic nie robi, i właśnie dlatego jest jednym z najczęściej uruchamianych programów w historii.

Klasyczne kopiowanie zbioru:

Sortowanie — na mainframie to osobny, wysoko zoptymalizowany produkt, a nie polecenie powłoki:

I kompilacja z uruchomieniem, przez gotową procedurę katalogowaną:

Kluczowe parametry, na które patrzy się przez lata:

- `DISP=(status, koniec-normalny, koniec-błędny)` — co ze zbiorem przed, po i w razie awarii. `(NEW,CATLG,DELETE)` czyta się: utwórz nowy, przy powodzeniu skataloguj, przy błędzie skasuj. To jest *transakcyjność na poziomie języka sterującego*.
- `SPACE=` — ile miejsca przydzielić, w śladach (TRK) lub cylindrach (CYL), z przydziałem pierwotnym i wtórnym.
- `DCB=` — format rekordu (`RECFM`: FB, VB, U), długość rekordu (`LRECL`) i rozmiar bloku fizycznego (`BLKSIZE`).

I tu dochodzimy do najważniejszej różnicy filozoficznej wobec Uniksa. W Uniksie plik *po prostu istnieje* i przyjmuje dowolne bajty. Na mainframie **trzeba z góry powiedzieć, jak zbiór jest zbudowany**: jakiej długości są rekordy, jak są zblokowane, ile miejsca zarezerwować. Zbiór to nie strumień bajtów, tylko *ustrukturyzowana kolekcja rekordów o zadeklarowanym formacie fizycznym*. To wygląda na uciążliwe, bo jest — ale w zamian system wie o danych wszystko i może je optymalizować, a narzędzia mogą je przetwarzać bez zgadywania.

JCL ma zasłużoną reputację wrogiego. Komunikaty potrafią być lakoniczne do granic („JCL ERROR”), pojedynczy przecinek w złym miejscu unieważnia zadanie, a diagnostyka nierzadko wskazuje objaw, nie przyczynę. Żart branżowy mówi, że JCL karze milczeniem.

Nazwy zbiorów i biblioteki

Nazwa zbioru ma **do 44 znaków** podzielonych kropkami na kwalifikatory po maksimum 8 znaków: `KONIO.PAYROLL.DANE.G0001V00`. Pierwszy człon — **HLQ** (High-Level Qualifier) — to zwykle identyfikator użytkownika lub kod projektu i służy też jako podstawa kontroli dostępu w RACF.

Nie ma katalogów w uniksowym sensie. Jest za to **PDS** (Partitioned Dataset) — zbiór-biblioteka zawierający nazwane *members*, czyli coś pomiędzy katalogiem a archiwum. Klasyczny PDS ma katalog o stałym rozmiarze i wymaga okresowego „compressu” po skasowaniach; nowszy **PDSE** tego problemu nie ma.

3270: terminal, który nie wysyła znaków

Terminal IBM 3278

IBM 3278 — „zielony ekran”. Nie wysyła pojedynczych znaków: użytkownik wypełnia formularz lokalnie, a dopiero ENTER przesyła całą zmodyfikowaną zawartość. Źródło: Wikimedia Commons.

To jest jeden z najważniejszych i najrzadziej rozumianych szczegółów całej platformy.

Terminal **IBM 3270** nie działa jak VT100. VT100 przy każdym naciśnięciu klawisza wysyła znak do hosta, a host odsyła echo — czyli każde uderzenie w klawiaturę generuje przerwanie i pracę procesora. **3270 jest terminalem blokowym.** Ekran (typowo 80×24, czyli 1920 bajtów) jest *buforowany lokalnie*, podzielony na **pola** z atrybutami: kolor, podświetlenie, ochrona przed edycją, znacznik „zmodyfikowano”. Użytkownik wypełnia formularz — *host o tym nic nie wie* — i dopiero naciśnięcie klawisza akcji (ENTER albo klawisz funkcyjny) wysyła **tylko zmienione pola**, jednym pakietem.

Konsekwencja jest gigantyczna i tłumaczy całą architekturę aplikacji mainframe'owych: **jeden host mógł obsłużyć tysiące terminali** przy minimalnym obciążeniu. Cytowany przykład z epoki: IBM 3033 z *szesnastoma megabajtami* pamięci obsługiwał do **17 500 terminali 3270** pod CICS-em. Kontrolery 3274/3174 dokładały kolejną warstwę multipleksowania.

Cena tego rozwiązania: nie da się zrobić edytora działającego znak po znaku jak `vi`. Stąd charakterystyczny rytm pracy — „wypełnij ekran, naciśnij Enter” — i stąd cały model programowania CICS, o którym w rozdziale 12.

TSO, ISPF i opcja 3.4

Terminal IBM 3270 z piórem świetlnym

Terminal 3270 z piórem świetlnym — wskaźnikiem, zanim upowszechniła się mysz. Źródło: Wikimedia Commons.

Interaktywna praca odbywa się przez **TSO** (Time Sharing Option) z nakładką **ISPF** — hierarchicznym systemem menu i paneli. Najczęściej wpisywana sekwencja w historii tej platformy to prawdopodobnie **„3.4”** — Data Set List Utility, czyli lista zbiorów. Do oglądania kolejek zadań i wydruków służy **SDSF**.

Tak wygląda ekran, przy którym spędza się tam całe dnie:

A tak — lista zadań w kolejce, oglądana w SDSF:

Automatyzację pisze się zwykle w **REXX-ie** — języku, który wyrósł na VM/CMS i jest na tej platformie tym, czym bash w Linuksie:

Zwróć uwagę na funkcję `SYSDSN` i na `ADDRESS TSO` — REXX ma wbudowany mechanizm kierowania poleceń do konkretnego środowiska (TSO, ISPEXEC, CICS), więc jeden skrypt może rozmawiać z kilkoma podsystemami naraz. To rozwiązanie, do którego PowerShell doszedł trzydzieści lat później.

Abendy: kody, które rozpoznaje się po zapachu

Awaryjne zakończenie zadania to **ABEND** (abnormal end), z kodem, który doświadczony programista czyta jak diagnozę:

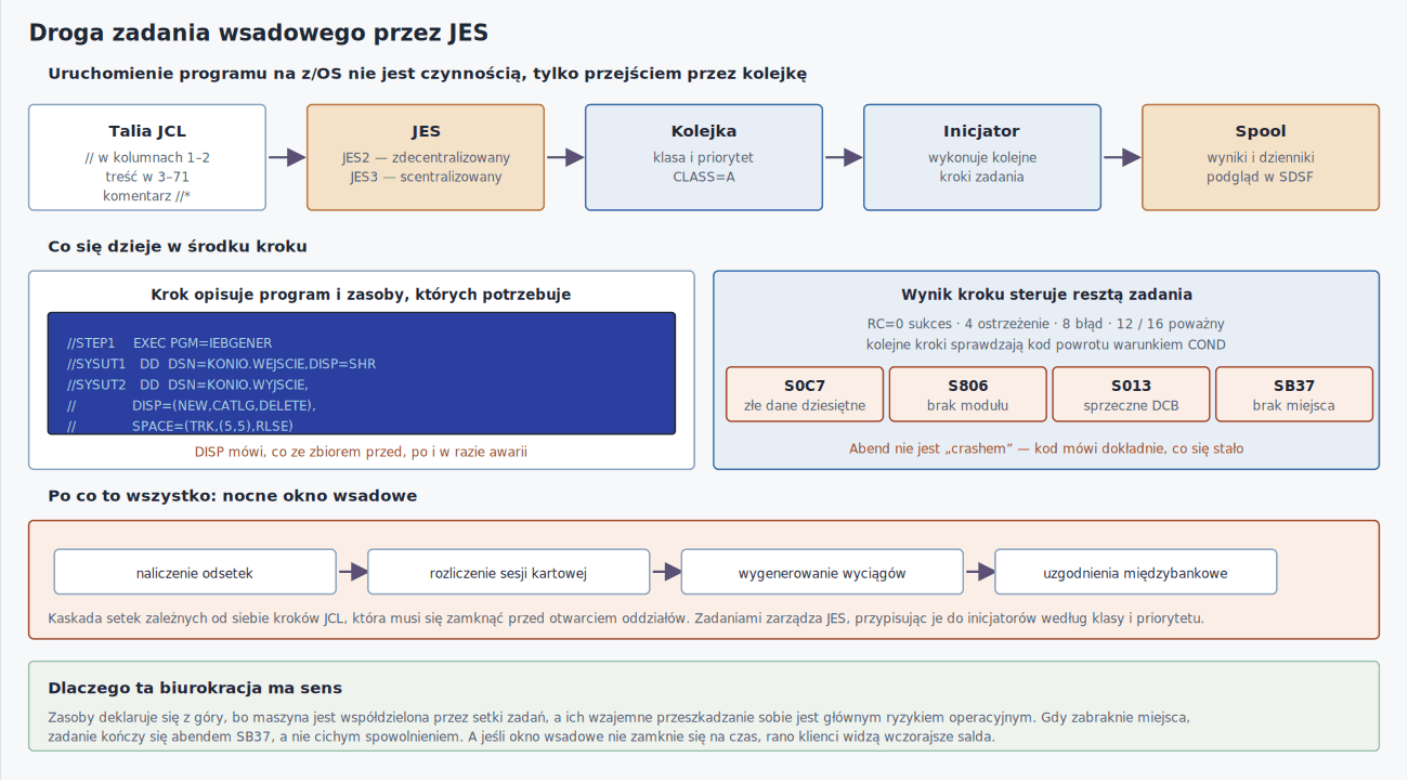
Rytm dnia: okno wsadowe

Czytnik kart IBM 3505

Czytnik kart IBM 3505. Zanim zadania wsadowe zaczęto wysyłać z terminala, „submit” oznaczał dosłownie położenie talii kart na podajniku. Źródło: Wikimedia Commons.

Na koniec pojęcie, które organizuje życie w każdym banku i każdej firmie ubezpieczeniowej: **nocne okno wsadowe**. Zadaniami zarządza **JES** (Job Entry Subsystem, w wariantach JES2 — zdecentralizowanym, i JES3 — scentralizowanym), przypisując je do *inicjatorów* według klasy i priorytetu, a wyniki trafiają na **spool**.

W nocy, gdy transakcje ucichną, rusza kaskada setek zależnych od siebie kroków JCL: naliczenie odsetek, rozliczenie sesji kartowej, wygenerowanie wyciągów, uzgodnienia międzybankowe. To musi się zamknąć przed otwarciem oddziałów. Jeśli okno wsadowe „nie zamknie się” na czas, rano klienci widzą wczorajsze salda. Stąd presja operacyjna, którą trudno przecenić — i stąd bierze się cała branża narzędzi do harmonogramowania zadań, o których świat webowy nigdy nie słyszał.



Rysunek 4. *Zadanie wsadowe nie jest uruchomieniem programu, tylko przejściem przez kolejkę z deklaracją zasobów, kodem powrotu i miejscem w harmonogramie — i właśnie dlatego jego opóźnienie widać rano na saldach klientów.*

Ściągawka: to samo zadanie w pięciu światach

Te same czynności w pięciu światach, obok siebie. Zestawienie jest wspólne dla całej serii i utrzymywane na stronie *Tabela systemów i porównanie poleceń*; kolumny stoją tam w stałej kolejności, niezależnie od tego, o którym systemie jest dany tekst.

Cztery obserwacje, które z tej tabeli wynikają wprost.

Po pierwsze, połowa komórek w kolumnie mainframe'owej to nie polecenia, tylko nazwy programów narzędziowych uruchamianych zadaniem wsadowym. `IEBGENER`, `IDCAMS`, `ICKDSF`, `ADRDSSU`, `DFSORT` — to nie są komendy powłoki, tylko moduły ładowalne, którym JCL podaje zbiory wejściowe i wyjściowe. Skopiowanie zbioru nie jest tu czynnością, lecz *zadaniem*: z deklaracją zasobów, kodem powrotu, wpisem w dzienniku i miejscem w kolejce. Brzmi to jak biurokracja i jest biurokracją — taką, której się chce, kiedy operacja dotyczy zbioru z saldami dwóch milionów rachunków.

Po drugie, utworzenie zbioru wymaga powiedzenia z góry, ile ma zajmować miejsca. W `DD` podaje się przydział pierwotny i wtórny (`SPACE=(TRK,(50,10))`), a gdy zabraknie — zadanie kończy się abendem `SB37`, a nie cichym spowolnieniem. To zupełnie inna umowa z systemem niż uniksowa: tam plik rośnie, dopóki starcza dysku. Tu zasoby deklaruje się i rezerwuje, bo maszyna jest współdzielona przez setki zadań, których wzajemne przeszkadzanie sobie jest głównym ryzykiem operacyjnym.

Po trzecie, tam gdzie inne systemy mają jedno polecenie, z/OS ma pełnoekranowy panel. Duża część codziennej pracy dzieje się nie w wierszu poleceń, lecz w ISPF i SDSF — aplikacjach 3270 z numerowanymi opcjami i klawiszami funkcyjnymi. „Opcja 3.4” nie jest komendą; to ścieżka w menu, która przez czterdzieści lat weszła administratorom w odruch bezwarunkowy. Blokowy terminal wymusił interfejs formularzowy, a interfejs formularzowy okazał się dla pracy masowej wygodniejszy niż wiersz poleceń — co jest jedną z ciekawszych rzeczy, jakie ta architektura ma do powiedzenia współczesnym projektantom narzędzi.

Po czwarte, REXX pojawia się w dwóch kolumnach — mainframe'owej i OS/2-owej. To nie zbieg okoliczności. Mike Cowlishaw napisał REXX-a w 1979 roku dla systemu VM/CMS, a IBM przeniósł go do OS/2 jako standardowy język skryptowy, bo firma myślała kategoriami dużych systemów nawet wtedy, gdy projektowała system na biurko. Jeśli szukać jednego, namacalnego dowodu na pokrewieństwo trzech światów opisanych w tej trylogii — to właśnie ten.

Fałszywi przyjaciele: te same słowa, inne znaczenia

Przybysza ze świata uniksowego najbardziej myli nie to, co na mainframie wygląda obco, lecz to, co wygląda znajomo:

> **Ciekawostka.** Każdy, kto pierwszy raz siada do ISPF, zadaje to samo pytanie: dlaczego klawisz F3 to „wyjdź”, a nie Escape? Odpowiedź jest archeologiczna. Terminal 3270 nie wysyłał znaków na bieżąco — żeby cokolwiek trafiło do komputera, trzeba było nacisnąć klawisz *akcji*: ENTER albo jeden z klawiszy PF. Escape nic nie wysyłał, więc nie mógł niczego oznaczać. Cała gramatyka klawiszy funkcyjnych, którą znamy z pecetów, wywodzi się z ograniczenia sprzętowego sprzed pięćdziesięciu lat.

Dane: VSAM, IMS i miejsce narodzin SQL

VSAM, czyli system plików, który jest bazą danych

Zanim pojawiły się relacyjne bazy danych, aplikacje potrzebowały indeksowanego dostępu do rekordów. Na mainframie daje go **VSAM** (Virtual Storage Access Method) — warstwa systemowa, nie osobny produkt. Cztery rodzaje zbiorów:

- **KSDS** (Key-Sequenced) — dostęp po kluczu, rekordy uporządkowane logicznie według klucza, osobny komponent indeksu i danych; to jest klasyczny ISAM;
- **ESDS** (Entry-Sequenced) — kolejność zapisu, dostęp przez RBA (względny adres bajtowy);
- **RRDS** (Relative Record) — dostęp po numerze slotu;
- **LDS** (Linear) — surowe bajty bez struktury rekordów; używa go między innymi Db2 na przestrzenie tabel.

Kluczowe pojęcia wewnętrzne to **Control Interval** (jednostka transferu I/O, np. 4 KB) i **Control Area** (grupa CI, jednostka podziału przy rozroście indeksu — stąd osławione „CI/CA splits”, zmora strojenia wydajności).

Zbiory VSAM tworzy się narzędziem **IDCAMS**, i to jest kolejny kawałek składni, który wygląda jak z innej planety:

Czytamy: rekordy po 200 bajtów, przydział 5000 rekordów plus po 1000 przy rozroście, klucz 10-bajtowy zaczynający się od pozycji 0, zbiór indeksowany. Myślniki na końcu linii to kontynuacja. Do tego dochodzą narzędzia: `ANALYZE` do diagnostyki struktury, `REPRO` do kopiowania, `EXPORT` / `IMPORT`.

IMS: hierarchia, która wzięła się z rakiety

IMS (Information Management System) ma najbardziej filmowy rodowód ze wszystkich produktów w tym tekście: powstał dla **programu Apollo**.

W 1963 roku NASA potrzebowała systemu zarządzania zmianami konstrukcyjnymi modułu dowodzenia. IBM i North American Rockwell zbudowały w zakładzie w Downey system **DATE** na komputerze IBM 7010, uruchomiony w 1965. Gdy maszyna się zestarzała, zaproponowano przeniesienie koncepcji na System/360. Formalna definicja powstała w **1966** — zespół liczył 12 osób z IBM, 10 z North American Rockwell i **3 z Caterpillar Tractor**. System, nazwany początkowo ICS/DL-I, zainstalowano **14 sierpnia 1968**, a IBM skomercjalizował go w 1969 jako **IMS/360**.

Problem był z natury hierarchiczny: *zestawienie materiałowe* rakiety Saturn V. Statek składa się z modułów, moduły z podzespołów, podzespoły z części, części ze śrub. Stąd **hierarchiczny model danych** (DL/I — Data Language/One): dane są **segmentami** ułożonymi w drzewo, połączonymi *fizycznymi wskaźnikami na dysku*, a nie kluczami obcymi.

Program deklaruje swój widok na bazę w **PCB** (Program Communication Block) — które segmenty widzi, w jakiej kolejności, z jakimi prawami — a zestaw PCB dla programu tworzy **PSB** (Program Specification Block).

Dlaczego IMS bywa szybszy od bazy relacyjnej? Bo nawigacja po drzewie to podążanie za wskaźnikami. Nie ma optymalizatora zapytań, nie ma złączeń, nie ma planu wykonania do wyliczenia w czasie działania — ścieżka dostępu jest ustalona na etapie generacji PSB. Dla z góry znanych, powtarzalnych wzorców dostępu (a rdzeń bankowy i rezerwacje lotnicze są dokładnie takie) to wygrywa z narzutem silnika SQL. Ceną jest całkowity brak elastyczności: zapytanie, którego nie przewidziano przy projektowaniu, jest praktycznie niewykonalne.

IMS nigdy nie umarł. W 1988 roku pracował w 7000 lokalizacji, w 2003 w ponad 95% firm z listy Fortune 1000, a aktualna linia to **IMS 15.x** z dokumentacją aktualizowaną w 2026 roku. W 2013 IBM demonstrował **100 000 transakcji na sekundę** na pojedynczym systemie IMS — to benchmark producenta w warunkach kontrolowanych, ale rząd wielkości mówi sam za siebie.

Gdzie naprawdę urodził się SQL

A teraz historia, która jest jednocześnie triumfem i najdroższą pomyłką biznesową w dziejach IBM.

W czerwcu 1970 roku **Edgar F. Codd**, matematyk pracujący w IBM, opublikował w *Communications of the ACM* artykuł **„A Relational Model of Data for Large Shared Data Banks”**. Zaproponował w nim, żeby dane traktować jako relacje matematyczne, a dostęp do nich opisywać *deklaratywnie* — mówiąc *co* chcemy dostać, a nie *jak* to znaleźć. To jest jedna z najważniejszych prac w historii informatyki.

IBM zbudował implementację: **System R**, w laboratorium w San Jose, w latach **1974–1979**. To tam **Donald Chamberlin** i **Raymond Boyce** zaprojektowali język zapytań o nazwie **SEQUEL** (Structured English QUEry Language). Nazwa musiała się zmienić, bo — jak głosi powszechnie powtarzana historia branżowa — była już zastrzeżonym znakiem towarowym brytyjskiej firmy lotniczej. Usunięto samogłoski i zostało **SQL**. *(Ta anegdota krąży wszędzie, ale nie udało się jej potwierdzić dokumentem pierwotnym — warto ją opowiadać z zastrzeżeniem.)*

I teraz pointa. IBM miał teorię (Codd), miał implementację (System R), miał język (SQL) i miał rynek. A na rynek z produktem wyszedł jako **trzeci**.

Dlaczego? Analizy wskazują na dwie przyczyny: IBM nie docenił siły koncepcji Codda i nie dał jej odpowiednich zasobów, a ponadto — to teza często powtarzana, choć trudna do udokumentowania — obawiał się kanibalizacji przychodów z IMS, który świetnie się sprzedawał. Klasyczny dylemat innowatora, opisany przez Claytona Christensena dwadzieścia lat później: firma nie wdraża własnego przełomu, bo zagraża on jej obecnemu, dochodowemu produktowi.

Dziś **Db2 for z/OS** (od kilku lat pisane z małym „b”) jest w wersji 13 i ma funkcję, której świat x86 zazdrości: **data sharing** — wiele instancji bazy na różnych systemach w klastrze czyta i zapisuje *te same* dane, ze spójnością wymuszaną sprzętowo. O tym w rozdziale 14.

CICS: oprogramowanie, którego nie widzisz

Jeśli mainframe jest niewidzialny, to **CICS** jest niewidzialny podwójnie. Prawie nikt spoza branży o nim nie słyszał, a jest to prawdopodobnie najintensywniej używany monitor transakcyjny w historii.

Geneza i model, który wyprzedził epokę

CICS (Customer Information Control System) powstawał od 1966 roku we współpracy z **Michigan Bell** — firma użyteczności publicznej potrzebowała dostępu do danych klienta w czasie rzeczywistym. Produkt zapowiedziano w 1968 jako „Public Utility Customer Information Control System”, a ogólną dostępność osiągnął **8 lipca 1969**. Napisany był w asemblerze — i do dziś jego rdzeń zawiera ogromne ilości kodu asemblerowego oraz PL/S, wewnętrznego dialektu PL/I używanego przez IBM do pisania systemów.

Kluczem jest model **pseudokonwersacyjny**, i warto go zrozumieć dokładnie, bo to jedna z najelegantszych odpowiedzi na ograniczenia sprzętowe, jakie wymyślono.

Problem: masz dziesięć tysięcy terminali i osiem megabajtów pamięci. Użytkownik siedzi przed ekranem i myśli przez trzydzieści sekund. Jeśli na czas tego myślenia utrzymujesz aktywne zadanie z całym jego kontekstem, przy tysiącu użytkowników maszyna się dusi.

Rozwiązanie CICS: **transakcja kończy się w chwili wysłania ekranu**. Program wykonuje `EXEC CICS SEND MAP`, po czym `EXEC CICS RETURN` z parametrem `TRANSID` i `COMMAREA` — i przestaje istnieć. Zwolniona zostaje pamięć, zwolnione zasoby. Gdy użytkownik naciśnie ENTER, CICS **uruchamia program od nowa** (nie wznawia!), odtwarzając stan z danych przekazanych w `COMMAREA` (w nowszych wersjach — w kanałach i kontenerach).

Innymi słowy: CICS wynalazł **bezstanową obsługę żądań z jawnie przekazywanym kontekstem** — czyli dokładnie ten model, który pół wieku później branża odkryła pod nazwami REST i serverless — nie dlatego, że był elegancki, tylko dlatego, że pamięć była droga.

Zwróć uwagę na ostatnie zdanie: `RETURN TRANSID('CUST') COMMAREA(...)` — „gdy użytkownik coś naciśnie, uruchom transakcję CUST i podaj jej to”. Cały stan sesji mieści się w strukturze przekazanej jawnie. Żadnych sesji po stronie serwera, żadnych sticky sessions, żadnego load balancera, który musi pamiętać, gdzie kto siedzi.

Jednostką organizacyjną jest **region CICS** — przestrzeń adresowa z/OS z własnym zestawem programów i zasobów. Wiele regionów koordynuje się przez **MRO** i **CICSPlex SM**, z równoważeniem obciążenia i routingiem transakcji — czyli, znowu, mikroserwisy przed mikroserwisami.

Aktualna wersja to **CICS Transaction Server for z/OS 6.3**, aktywnie łatana (poprawki z 2026 roku), ze wsparciem OpenTelemetry i integracją z Visual Studio Code.

Kolejka komunikatów: MQ

Do kompletu warto wymienić **MQSeries**, wydany w grudniu 1993, przemianowany na WebSphere MQ (2002), a potem na **IBM MQ** (2014). Asynchroniczna, transakcyjna kolejka komunikatów z gwarancją dostarczenia, dostępna dziś na kilkudziesięciu platformach — koncepcyjny przodek Kafki, RabbitMQ i SQS, urodzony na mainframie.

Ile tego jest — i ile z tego wiemy na pewno

Tu trzeba być ostrożnym, bo prawie wszystkie liczby o skali mainframe'ów pochodzą od ich producenta.

Materiały IBM podają: „mainframe'y obsługują 68% światowych produkcyjnych obciążeń IT”, „90% transakcji kartowych”, „92 ze 100 największych banków”, „23 z 25 największych detalistów”, „7,7 biliona dolarów w płatnościach kartowych rocznie”. Ta ostatnia liczba jest o tyle ciekawsza, że IBM podaje przy niej przypis do zewnętrznego *The Nilson Report* (2016) — ale bez metodologii.

Żadna z tych liczb nie pochodzi z niezależnego audytu. Próby dotarcia do źródła pierwotnego dla najczęściej powtarzanych („68% obciążeń”, „70% firm z Fortune 500”) kończą się w blogach partnerów handlowych i firm sprzedających usługi migracji z mainframe'a — czyli u podmiotów, które też mają interes w wyolbrzymianiu skali, tylko z przeciwnego powodu. To są statystyki, które

żyją własnym życiem.

Uczciwsze jest więc sformułowanie jakościowe, którego nikt nie kwestionuje: **w sektorach, gdzie pojedyncza transakcja musi być poprawna, szybka i rozliczalna — bankowość, karty płatnicze, ubezpieczenia, rezerwacje, administracja podatkowa — mainframe pozostaje domyślną platformą rdzenia systemu.** Bez procentów.

Revision #2
Created 2026-11-03 00:54:58 UTC by Tomasz Konieczny
Updated 2026-11-03 06:29:04 UTC by lektor