FAKULTA MECHATRONIKY,
INFORMATIKY A MEZIOBOROVÝCH
STUDIÍ TUL

Diplomová práce

Nástroje pro monitoring síťových zařízení

Studijní program:	N0613A140028 Informační technologie
Studijní obor:	Výpočetní systémy
Autor práce:	**Bc. Jan Veverka**
Vedoucí práce:	Ing. Jan Kraus, Ph.D. Ústav mechatroniky a technické informatiky

Liberec 2024

FAKULTA MECHATRONIKY,
INFORMATIKY A MEZIOBOROVÝCH
STUDIÍ TUL

Zadání diplomové práce

Nástroje pro monitoring síťových zařízení

Jméno a příjmení:	**Bc. Jan Veverka**
Osobní číslo:	M22000035
Studijní program:	N0613A140028 Informační technologie
Specializace:	Výpočetní systémy
Zadávající katedra:	Ústav mechatroniky a technické informatiky
Akademický rok:	2023/2024

Zásady pro vypracování:

1. Proveďte analýzu dostupných a otevřených monitorovacích systémů, stručně popište jejich architekturu, vlastnosti a principy fungování.
2. Pro praktickou část práce vyberte vhodnou kombinaci monitorovacích systémů s ohledem na zabezpečení, rozšiřitelnost, distribuovatelnost a škálování celého navrženého řešení pro použití v prostředí s větším počtem sledovaných zařízení.
3. Své návrhy implementujte a na funkčním vzorku demonstrujte základní funkce, rozhraní a vlastnosti vytvořeného systému.
4. V závěru popište dosažené výsledky a diskutujte možnosti využití výsledků v praxi.

Rozsah grafických prací:	dle potřeby dokumentace
Rozsah pracovní zprávy:	40 až 50 stran
Forma zpracování práce:	tištěná/elektronická
Jazyk práce:	čeština

Seznam odborné literatury:

[1] MOTA, Levi Costa; MORENO, Edward David; RIBEIRO, Admilson Lima. A comparative analysis of protocols for IoT network management. In: Proceedings of the Euro American Conference on Telematics and Information Systems. 2018. p. 1-5.

[2] ZHAO, Yuyu, et al. Snapshot for Power Grids IoT: Adaptive Measurement for Resilience Intelligent Internet of Things. IEEE Internet of Things Journal, 2023.

[3] SHAN, Yang Guo, et al. Research on Monitoring of Information Equipment Based on Zabbix for Power Supply Company. In: 2021 3rd International Conference on Applied Machine Learning (ICAML). IEEE, 2021. p. 487-491.

[4] DIAZ-CACHO, Miguel, et al. Educational Test-bed for Maintenance 4.0. In: 2022 IEEE Global Engineering Education Conference (EDUCON). IEEE, 2022. p. 1310-1315.

Vedoucí práce:	Ing. Jan Kraus, Ph.D. Ústav mechatroniky a technické informatiky
Datum zadání práce:	12. října 2023
Předpokládaný termín odevzdání:	14. května 2024

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

doc. RNDr. Pavel Satrapa, Ph.D.
garant studijního programu

V Liberci dne 12. října 2023

Prohlášení

Prohlašuji, že svou diplomovou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Jsem si vědom toho, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS/STAG se shoduje s textem tištěné podoby práce.

Beru na vědomí, že má diplomová práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.

Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

7. května 2024 Bc. Jan Veverka

Nástroje pro monitoring síťových zařízení

Abstrakt

Tato diplomová práce se zabývá návrhem a ukázkovou implementací pokročilé architektury pro monitorování sítí zařízení internetu věcí. Práce zahrnuje analýzu existujících open-source monitorovacích nástrojů a definování kritérií pro architekturu, která by měla splňovat požadavky na bezpečnost, distribuovatelnost, škálovatelnost a rozšiřitelnost. Výsledkem práce je důsledně navržená architektura, která je schopna efektivně fungovat v rozsáhlých systémech díky důrazu na požadované konkrétní vlastnosti. V závěru práce je vyzdvižen potenciál navrhované architektury k využití v širokém spektru reálných aplikací a přínos pro kompetence autora v komplexním prostředí internetu věcí.

Klíčová slova: otevřené monitorovací nástroje, architektura monitorovacího systému, internet věcí, Prometheus, Grafana

Tools for monitoring of network appliances

Abstract

This thesis addresses the design and implementation of an advanced architecture for monitoring networks of Internet of Things devices. The thesis entails an analysis of existing open-source monitoring tools and a definition of criteria for the architecture to meet the requirements of security, distributability, scalability and extensibility. The thesis result is a thoroughly designed architecture that is able to operate effectively in large-scale systems thanks to the emphasis on the specific features required. The thesis concludes by demonstrating the potential of the proposed architecture to be used in a wide range of real-world applications and the contribution to the author's competence in the complex Internet of Things environment.

Keywords: open-source monitoring tools, monitoring system architecture, Internet of Things, Prometheus, Grafana

Poděkování

Rád bych poděkoval všem, kteří přispěli, přímo či nepřímo, ke vzni-
-ku této práce. Upřímně děkuji vedoucímu práce, rodině a kamará-
dům.

Obsah

Seznam obrázků

Seznam zkratek

TUL	Technická univerzita v Liberci
FM	Fakulta mechatroniky, informatiky a mezioborových studií Technické univerzity v Liberci
IoT	internet věcí („Internet of Things“)
AWS	webové služby Amazon („Amazon Web Services“)
MQTT	síťový protokol („MQ Telemetry Transport“)
TCP	protokol řízení přenosu („Transmission Control Protocol“)
HTTP	internetový protokol pro komunikaci s webovými servery („Hypertext Transfer Protocol“)
HTTPS	zabezpečený internetový protokol pro komunikaci s webovými servery („Hypertext Transfer Protocol Secure“)
JSON	JavaScriptový objektový zápis („JavaScript Object Notation“)
NaN	hodnota, která nereprezentuje číslo („Not a Number“)
SNMP	jednoduchý protokol pro síťový management („Simple Network Management Protocol“)
JMX	Java technologie pro správu a monitorování aplikací („Java Management Extensions“)
SSH	zabezpečený komunikační protokol („Secure Shell“)

1 Úvod

V současné době zažívá mnoho technologií výrazný pokrok. Jedna z takovýchto technologií je internet věcí, který umožňuje spojovat a efektivně využívat velké množství jednotlivých zařízení. Konkrétní zařízení v systému plní určité konkrétní role a v některých systémech může být velmi přínosné tato zařízení určitým způsobem monitorovat. Platí to jak pro zařízení určená k samotnému sběru dat, tak pro zařízení, jejichž primárním cílem není generování dat. Tato práce si neklade za cíl popsat jeden konkrétní systém a v něm „na míru“ implementovat monitorovací systém, ale vytvořit efektivní, poměrně obecnou a adaptabilní architekturu zaměřenou na monitoring systémů s velkým počtem zařízení. Takové systémy mají totiž ve většině případů velmi podobné požadavky na vlastnosti monitorovacího systému. Tato práce se zaměřuje na definování daných vlastností a vytvoření komplexní architektury, která vedle navržených vlastností bude splňovat požadavky na bezpečnost, distribuovatelnost, škálovatelnost a rozšiřitelnost. Ve výsledku bude architektura implementována v rámci ukázkového řešení, respektive simulovaného systému, aby byly demonstrovány funkce, vlastnosti a rozhraní dané architektury. Teoretická část je zaměřená na výběr konkrétních aplikací použitých ve výsledné architektuře a na definování konkrétních vlastností, které by výsledná architektura měla mít. Praktická část je zaměřená na realizaci dané architektury v konkrétním případu užití, kde je vytvořen simulovaný systém, pro který je vhodné implementovat monitorovací architekturu.

2 Internet věcí

2.1 Síťová zařízení

Počet síťových zařízení na světě neustále roste. To je dáno nejen klasickými zařízeními typu mobilní telefon nebo počítač, ale hlavně počet síťových zařízení roste kvůli čím dál většímu počtu zařízení IoT (zkratka z anglického „Internet of Things“, česky internet věcí). Podle článku [1] od společnosti Statista se počet globálních zařízení internetu věcí ztrojnásobí z 8,74 miliardy v roce 2020 na více než 25,4 miliardy v roce 2030. Součástí této diplomové práce je analýza dostupných a otevřených monitorovacích systémů, protože při velkém počtu zařízení je vhodné mít ideální monitorovací systém kvůli přehlednosti a snadnému provádění diagnostiky. Pojem síťové zařízení je velmi široký, jedná se totiž o všechna zařízení připojená do sítě. Těžko ale zobecňovat konkrétní monitorovací systémy pro tak širokou škálu různých druhů zařízení. V této práci se předpokládá monitorování standardních zařízení pro použití v IoT systémech. Taková zařízení se typicky vyznačují systémem Linux a jednoduchým hardwarem, který je ale dostatečně rychlý na jednoduché zpracování dat a odesílání daných dat po síti.

2.2 Škálovatelnost a distribuovatelnost systému

Teoreticky může být v systému nekonečně mnoho IoT zařízení, které je třeba monitorovat. Proto je škálovatelnost velmi důležitá vlastnost monitorovacího systému. Škálovatelnost vyjadřuje schopnost zvládnutí nárůstu objemu práce. V praxi to znamená, že každé přidané zařízení do IoT systému zvyšuje síťový provoz, dále odesílaná data musí nějaké další zařízení přijímat, a aby mělo monitorování nějaký smysl, tak daná data i nějakým dalším způsobem zpracovávat. Škálovatelnost se dělí na horizontální a vertikální. Vertikální škálování je škálování založené na zvyšování výkonu jednoho uzlu, v praxi serveru. To má své výhody i nevýhody. Je to poměrně snadné

řešení, zvýšit výkon u slabších strojů není problém. Velkým problémem je ale to, že se výkon nedá zvyšovat donekonečna, takže ve výsledku se nutně musí buď přidat další uzel, nebo přestat škálovat daný systém. Horizontální škálovatelnost znamená přidání dalších výpočetních uzlů, a tudíž snížení zátěže každého z nich. To má velkou výhodu v teoreticky neomezeně škálovatelném systému, ale v praxi to naráží na problémy spojené s paralelním zpracováním dat. Proto je vhodné při návrhu každého systému brát v potaz jeho distribuovatelnost.

Distribuovatelnost je vlastnost systému pracovat na více uzlech zároveň kvůli snížení požadavků na výkon každého uzlu. Zvenku se tedy systém chová jako jedna celistvá aplikace, ale ve skutečnosti je rozdělena na více malých podúloh, kde každou z nich může vykonávat jiný výpočetní uzel. Ne každý systém ale takové rozdělení podporuje, protože distribuovaný systém musí řešit určité problémy, které se s distribucí pojí.

Distribuované systémy jsou prakticky zvláštní typ paralelních systémů. Paralelní systémy jsou takové systémy, kde se na výpočtu podílí více procesorových jader či případně více procesorů jako takových. Nevýhodou distribuovaného systému je například větší náročnost na hardware, protože hardware musí řešit kromě samotných výpočtů i úlohy spojené se samotnou distribucí jednotlivých částí systému. Další z nevýhod je vyšší zatížení samotné sítě a bezpečnost, která musí být pečlivě ošetřena z důvodu snadného přístupu ke sdíleným datům v dané síti.

2.3 Time-series data

Time-series data jsou data, která jsou ukládána jako chronologicky indexovaná řada. Typicky se jedná o data, mezi jejichž sběrem uběhne stejně dlouhá doba. Jenže data sbíraná při monitoringu zařízení lze většinou rozdělit do dvou skupin. Do první skupiny patří data sbíraná periodicky za účelem zjištění průběhu sledované veličiny. Mezi tato data může patřit zátěž procesoru, teplota zařízení nebo okolí, spotřeba paměti a podobné informace. Druhá typická skupina dat jsou data, která jsou generovaná nějakou událostí nebo při nějaké události. Taková data nebývají typicky generována periodicky, ale i taková data mohou profitovat z uložení jako time-series data. Výhodou time-series dat je jejich intuitivní a snadná vizualizace průběhu v čase díky tomu, že samotná data obsahují časovou řadu.

3 Monitorovací systémy

3.1 Stručná historie monitorovacích systémů

Dnešním moderním monitorovacím systémům předcházely systémy, které se zaměřovaly především na specifické úkoly a často byly omezeny svými technologickými možnostmi. Dnes lze snadno monitorovat pomocí jednoho moderního systému jak informace o síťových přenosech a s tím souvisejících náležitostech, tak i data z konkrétních senzorů v rámci sběru dat z IoT zařízení. Historicky to tak ale nebylo a při návrhu systému bylo nezbytné podrobně specifikovat monitorovaná zařízení.

V průmyslu bylo často potřeba monitorovat průběhy dějů, které na sebe navazovaly, například v rámci výrobní linky, takže vznikaly oddělené ostrovní jednoúčelové monitorovací systémy. S postupem času se takové sítě mohly díky rozmachu počítačových sítí propojovat do větších celků a v dnešní době mohou být propojené do komplexních IoT sítí. V současnosti je oproti historickým řešením největší rozdíl v použití efektivních protokolů pro komunikaci, databází typu time-series a pokročilých, snadno škálovatelných a implementovatelných univerzálních nástrojů.

Pro monitoring sítí se v historii využívaly nástroje často omezené z hlediska škálovatelnosti, flexibility a integrace. Mezi takové nástroje patřil například Syslog, který definoval standard pro posílání logů zařízení pomocí zpráv do centrálního serveru. Proto je dnes již nedostačující, navíc centralizované zpracování logů bylo těžko škálovatelné a pro velké objemy logovaných dat mohlo docházet k problémům s výkonem. Syslog také neumožňuje posílání zpráv v modernějším formátu, jako je JSON, a tudíž jejich analýza je složitější. Zabezpečení je také na nižší úrovni než v moderních monitorovacích systémech, které mají větší možnosti zabezpečení jak přenosu, tak ukládání potenciálně citlivých dat, což logy mohou být.

Moderní monitorovací systémy bývají robustnější a mají obecně lepší vlastnosti v rámci distribuovatelnosti, škálování, zabezpečení, implementace a ukládání sbíraných dat.

Následující otevřené monitorovací softwary byly vybrány a dále zkoumány na základě osobní zkušenosti autora s některými z nich a na základě online zdrojů [2], [3], [4] a [5].

3.2 Zabbix

Zabbix je otevřený monitorovací software s širokou škálou použití. Pomocí Zabbixu je možné monitorovat například sítě, servery, cloudové služby, kontejnerové aplikace, databáze, služby nebo aplikace. Prakticky je tedy možné monitorovat téměř vše, co dává smysl monitorovat. Kromě monitorování umí Zabbix sbíraná data vizualizovat a vytvářet upozornění pro uživatele.

Samotný sběr dat ze zařízení může probíhat dvěma způsoby. První způsob využívá tzv. agenta, který se nainstaluje na monitorované komponentě systému a má přístup k požadovaným datům. Agent následně odesílá data do nadřazeného serveru, kde probíhá zpracování dat. Druhý způsob je pouze jednoduchá kontrola konkrétní služby bez agenta na monitorované komponentě, prakticky Zabbix posílá dotazy na konkrétní port některých služeb a odpovědi dané služby jsou námi monitorovaná data. Tato varianta funguje pouze pro vybrané služby a protokoly jako jsou například VMware, SSH (zkratka z anglického „Secure Shell“), JMX (zkratka z anglického „Java Management Extensions“), SNMP (zkratka z anglického „Simple Network Management Protocol“) nebo Telnet.

Jedna z hlavních výhod, ale zároveň i nevýhod Zabbixu je jeho velká konfigurovatelnost. Konfigurují se nejen monitorovaná zařízení a jejich skupiny, ale i pokročilejší záležitosti, jako jsou pravidla pro sběr konkrétních monitorovaných dat nebo triggery, které na základě sbíraných dat definují například stav systému. Dále se dají konfigurovat dashboardy, díky kterým můžeme snadno vizualizovat, a proto lépe kontrolovat hodnoty sbíraných dat. Na dashboardy se dají umisťovat grafy, mapy, síťové mapy nebo logy akcí a podobné užitečné nástroje. Důležitá je také správa oprávnění jednotlivých uživatelů a skupin, lze omezit přístup k administrativní části frontendu, provádění jednotlivých akcí, přístup k monitorovaným zařízením a používání určitých API volání. Podle zdroje [6] je výhoda Zabbixu flexibilita při organizaci monitorovacích dat, konfigurovatelnost a škálovatelnost.

3.3 Node-RED

Node-RED je programovací nástroj pro jednoduché vizuální programování aplikací. Podle zdroje [7] je založený na takzvaných uzlech, angl. nodes, a propojení mezi nimi. Programování je tzv. flow-based a funguje na principu jednotlivých uzlů, které mají definované chování. Do každého z nich vstupují data, ta se mohou nějakým způsobem zpracovat a poté se předávají do dalšího uzlu. Samotné programování je velmi vizuálně přehledné, připomíná blokový diagram, což je pro běžného uživatele velmi přívětivé. Uživatel, který programuje tyto uzly, tak nemusí být programátor a nemusí ve většině případů psát vlastní kód, stačí správně poskládat bloky, správně je propojit a definovat jejich chování.

Node-RED se skládá z run-time prostředí založeném na Node.js a webové aplikace, v níž se pomocí zmiňovaných uzlů a jejich propojení složí aplikace, kterou lze následně spustit. Velká komunitní výhoda je možnost sdílet aplikace jako soubory JSON, ve kterých jsou popsány jednotlivé datové toky jako propojení jednotlivých uzlů.

Velká výhoda nástroje Node-RED je při integraci do větší aplikace. To je také důvod, proč je v této diplomové práci Node-RED zmiňovaný. Při práci s větším množstvím dat, které velké IoT systémy mohou generovat, je velká pravděpodobnost, že bude nutné provádět s některými daty netriviální operace, jako je například přeformátování, a na takové operace nemusí samotný monitorovací software stačit. Proto může být vhodné včlenit do celého systému nástroj Node-RED nebo podobný software. Node-RED se hodí k doplnění jiného monitorovacího softwaru díky snadnému programování a snadné integraci v rozsáhlejší architektuře, které je dosaženo širokou škálou vstupních a výstupních uzlů ze systému. Samotný nástroj Node-RED lze využít i samostatně pro tvorbu dashboardů, ale není to ideální řešení pro komplexnější systémy kvůli nepřehlednosti při větším počtu uzlů.

3.4 Prometheus

Prometheus je otevřený monitorovací systém, který efektivně využívá strukturu time-series dat. Zdroj [8] uvádí, že Prometheus je sada nástrojů pro monitorování a upozorňování původně vytvořená ve společnosti SoundCloud. Od svého vzniku v roce 2012 si Prometheus osvojilo mnoho společností a organizací a projekt má

velmi aktivní komunitu vývojářů a uživatelů. Nyní je samostatným otevřeným projektem a je udržován nezávisle na jakékoli společnosti. Z důvodu zdůraznění této skutečnosti a vyjasnění struktury správy projektu se systém Prometheus připojil v roce 2016 k nadaci Cloud Native Computing Foundation jako druhý hostovaný projekt po Kubernetes. Jak již bylo zmíněno, Prometheus sbírá a ukládá data ve formě time-series dat, což znamená, že sbíraná data jsou ukládána spolu s časem do časové řady, volitelně spolu s označeními, tzv. labely.

Vývojáři ve zdroji [8] popisují jeho hlavní funkce tak, že se jedná o multidimenzionální datový model, který kromě samotných time-series dat také obsahuje Prometheus Query Language (PromQL), což je dotazovací jazyk, který je vytvořen za účelem výběru a agregace time-series dat v reálném čase.

Prometheus v základní konfiguraci nepracuje s distribuovaným úložištěm, což se může zdát jako velká nevýhoda v kontextu škálovatelnosti a distribuovatelnosti celého systému, ale Prometheus obsahuje mechanismy, kterými lze toto omezení velmi dobře vyřešit. Mezi tyto mechanismy patří federace, která zavádí více instancí Prometheа v takové hierarchii, že jedna instance shromažďuje vybrané metriky z jí podřazených instancí Prometheа. Další mechanismus pro snadné škálování a distribuovatelnost je použití externího datového úložiště. V dokumentaci Prometheа nalezneme zdokumentované postupy pro interakci Prometheа s mnoha různými databázovými systémy, jako je například AWS (zkratka z anglického „Amazon Web Services“, česky webové služby Amazon) Timestream, Elasticsearch nebo Grafana Mimir. Prometheus dokáže data vizualizovat, ale častější použití je ve spojení s Grafanou, protože ta je pro tvorbu dashboardů a vizualizací vhodnější. V prostředí IoT a monitoringu je Prometheus celkově velmi dobrý software díky konfigurovatelnosti, velmi dobré škálovatelnosti a relativně nízkým hardwarovým nárokům.

3.5 Grafana

Grafana je otevřený software, jehož hlavní využití spočívá ve vizualizaci dat. Kromě samotné vizualizace ho lze využít pro jednoduchou analýzu. Do Grafany lze také implementovat externí rozšíření, např. pro vizualizaci dat s geografickou polohou na mapách. Velká výhoda Grafany je to, že pro vizualizovaná data nemá svou konkrétní vlastní databázi, ale přidávají se již existující zdroje prakticky bez zásahu do nich. Proto je to velmi silný nástroj, kterým lze doplnit již existující architekturu pro snadnou vizualizaci dat. Tato vlastnost výrazně zvyšuje flexibilitu využití Grafany.

Velmi dobře funguje v kombinaci s již zmiňovaným systémem Prometheus, a proto se tyto dva softwary často používají spolu. Data lze získávat a vizualizovat i z databází, jako jsou PostgreSQL, MySQL, Elasticsearch, InfluxDB a další. Nejvhodnější pro snadnou vizualizaci průběhu dat v závislosti na čase jsou data ve formátu time-series, a to kvůli snadným operacím s nimi v prostředí Grafany. Jednoduchá agregace dat je snadno implementovatelná, Grafana poskytuje základní nástroje pro práci s proměnnými a s dotazy, což umožňuje jednoduše vytvářet složitější interaktivní vizualizace ve formě jednotlivých panelů, které se umisťují na dashboardy. Grafana je velmi často používaný, pokročilý vizualizační nástroj, který je použitelný v mnoha různých komplexních systémech.

3.6 Elastic Stack

Elastic Stack je soubor otevřených nástrojů, které spolu fungují jako efektivní architektura pro sběr, vyhledávání, vizualizaci a analýzu dat v reálném čase. Jedná se o oblíbené a efektivní řešení pro komplexní systémy. Zcela zjevná výhoda těchto nástrojů je jednoznačná architektura a jednoduchá provázatelnost jednotlivých součástí systému. Celý ELK Stack má také jednoduché nasazování, protože může fungovat jednoduše jako cloudová služba Elastic Cloud, kde si zákazník vybírá jen poskytovatele cloudu. Toto řešení je snadné a nevyžaduje složitou konfiguraci, ale cenově nemusí být nejefektivnější už jenom proto, že zákazník platí nejen za servery. Elastic Cloud se platí měsíčně a má několik verzí podle ceny a vlastností. Jednotlivé cenové varianty se podle zdroje [9] liší převážně konfigurovatelností, podporou pokročilých bezpečnostních funkcí, možnostmi reportů a upozornění a v neposlední řadě nástroji pro strojové učení.

Základní součásti Elastic Stacku jsou nástroje Elasticsearch, Logstash a Kibana. Kromě nich existují i další nástroje, ale toto jsou ty zásadní z nich. Zdroje [10] a [11] popisují jednotlivé části následovně. Elasticsearch je distribuovaný nástroj, který se používá pro vyhledávání a analýzu. Jeho princip spočívá v ukládání dat shromážděných z ostatních součástí Elastic Stacku. Data se do něj dostávají přes API nebo pomocí nástroje pro odesílání/příjem dat, jako je například Logstash, což je další zmiňovaná součást Elastic Stacku. Data se posílají ve formátu JSON, ke kterému se automaticky přiřadí vyhledatelná reference do indexu clusteru. Následkem toho se daný dokument pomocí API Elasticsearch dá snadno vyhledat a použít pro požadovaný záměr. Logstash je otevřený nástroj pro efektivní práci s daty. Jeho hlavní

funkce jsou příjem, jednoduchá transformace a odeslání dat do další části systému, většinou do již zmiňovaného Elasticsearch. Příjem dat je jeho silná stránka, protože je uzpůsobený pro sběr dat z mnoha různých zdrojů. Díky filtrům a pluginům lze rozumně zpracovat data prakticky nezávisle na jejich zdroji a typu. Pluginů je na GitHubu dostupných přes 200, takže v případě nějakého více či méně standardního datového zdroje je pravděpodobné, že někdo už zpracování dat vymyslel a sdílel daný otevřený plugin.

3.7 Nagios a Icinga

Nagios je další otevřený monitorovací systém. Je primárně určený k automatickému monitoringu počítačových sítí a souvisejících služeb převážně na zařízeních s Linuxem. Jedná se o velmi starý nástroj, který je ale dodnes udržován a aktualizován. Předchůdce Nagiosu NetSaint byl vydaný podle zdroje [12] v roce 1999 a v roce 2002 byl projekt kvůli problémům s právy přejmenován na Nagios. Podle oficiálního webu [13] umožňuje organizacím identifikovat a řešit problémy IT infrastruktury předtím, než ovlivní kritické procesy. Je také navržený s ohledem na škálovatelnost a flexibilitu. Mezi jeho hlavní funkce patří monitoring, výstrahy určené pro konkrétní skupiny uživatelů, reporting, snadné plánování a údržba systému. Funkce systému se dají snadno rozšířit pomocí rozsáhlé existující knihovny pluginů. Ta obsahuje přes 4 000 pluginů, které jsou vyvíjené komunitou.

Se softwarem Nagios souvisí Icinga, to je otevřený monitorovací systém, který vznikl v roce 2009 jako fork Nagiosu. Jeho výhodou je například kompatibilita pluginů s Nagiosem. Vývojáři Icingy se celkově snaží zachovat co největší zpětnou kompatibilitu a také systémy dále rozvíjejí především v oblasti monitorování firemní IT struktury.

3.8 OpenNMS

OpenNMS je další otevřený software pro správu a monitoring sítě. Nejsilnější stránkou OpenNMS je velký důraz na distribuovatelnost a škálovatelnost. Jedná se o komplexní monitorovací nástroj, který má velký potenciál pro monitoring sítí. Systém umožňuje automatickou detekci nových síťových zařízení a podporuje sběr dat pomocí několika různých protokolů, jakou jsou SNMP, HTTP (zkratka z anglického

„Hypertext Transfer Protocol“), JMX a další. Podpora REST API a Kafka je ideální pro komunikaci s dalšími komponentami v rámci větší architektury. Kafka je distribuovaná platforma pro komunikaci založenou na událostech a REST API se používá k předávání dat mezi aplikacemi. Vzhledem k vlastnostem je OpenNMS vhodný spíše pro monitoring IT infrastruktury.

3.9 Požadované vlastnosti systému pro monitoring síťových zařízení

Každý monitorovací systém bude mít více či méně rozdílné požadavky na monitorovací software. Jsou ale určité obecné vlastnosti, které je vhodné brát v potaz u většiny monitorovacích systémů. Následující vlastnosti jsou podle mého názoru velmi důležité pro správné fungování monitorovacích systémů.

- schopnost sbírat data z různých zdrojů, podpora různých protokolů a datových formátů
- automatická objevování nových zařízení v síti
- vizualizace dat
- agregace dat
- analýza dat a predikce trendů
- generování upozornění, efektivní nastavení thresholdů
- integrace s externími systémy - upozornění pomocí SMS, email, Slack
- automatizace reakce na specifické události
- API pro integraci s externím softwarem
- zabezpečení frontendu, autentizace a autorizace uživatelů
- rozdělení uživatelských práv a rolí
- interaktivita frontendu
- logování historie změn v systému, dohledatelnost operací
- monitorování nákladů spojených s vlastním provozem
- integrace dat z vlastních aplikací a hardwarových zařízení

3.10 Případ užití – monitoring farmy 3D tiskáren

Typickým případem užití monitorovacího systému je farma 3D tiskáren, protože sbírat a vyhodnocovat určitá data je velmi vhodné během tisku i mimo něj. Kvalitu tisku totiž mimo jiné velmi ovlivňuje nejen teplota samotné trysky tiskárny, ale i teplota vyhřívané podložky tiskárny. V závislosti na materiálu, ze kterého 3D tiskárna tiskne, jsou tiskárny buď otevřené, nebo jsou uzavřené v boxu. V obou variantách je potřeba kontrolovat teploty. Pokud je tiskárna uzavřená v boxu, tak lze předpokládat tisk z materiálů, jako jsou například ABS nebo ASA, které nutně potřebují vysokou a především stálou teplotu v tiskárně. Otevřená tiskárna zase je náchylná na průvan, respektive náhlý pokles teploty. Simulovaná tiskárna by proto měla odesílat především data o teplotě trysky a podložky, status tiskárny, procenta dokončení aktuální tiskové úlohy a váhu zbývající tiskové struny. Monitorovací systém by měl ideálně upozorňovat na potřebu jejího doplnění, aby se předešlo přerušení tisku kvůli vyčerpání tiskové struny.

4 Praktická implementace

Pro praktickou implementaci jsem vybral kombinaci tří základních komponent monitorovacího systému. První část architektury bude aplikace Prometheus, která bude použita pro sběr dat. Sbíraná data je nutné dále zpracovávat a vizualizovat. Jako komponentu pro vizualizaci a jednoduchou analýzu dat jsem vybral aplikaci Grafana, která je jednou z nejlepších otevřených aplikací pro vizualizace. Třetí základní komponenta je aplikace Node-RED, která bude použita pro více účelů, které souvisí s provozem, rozšiřitelností a univerzálností celé architektury.

4.1 Výběr softwaru na základě požadovaných vlastností

Tato část se věnuje důvodům pro výběr základních komponent Prometheus, Grafana a Node-RED. Tyto základní komponenty byly vybrány na základě definovaných požadovaných vlastností monitorovacího systému. Schopnost sbírat data z různých zdrojů a podporovat různé protokoly a datové formáty je první důležitá vlastnost, kterou má Prometheus na velmi vysoké úrovni díky tzv. exportérům. Ty figurují právě ve sběru dat, která nejsou tak snadno získatelná. Ve zdroji [14], dokumentaci Promethea, jich nalezneme celou řadu, od exportérů z konkrétních databází (např. MongoDB exporter nebo MySQL router/server exporter) přes exportéry související s hardwarem (např. NVIDIA GPU exporter nebo Netgear Router exporter) po nejrůznější API exportéry. Tyto exportéry vyvíjí třetí strany a každý si může vytvořit svoje podle pravidel popsaných v dokumentaci Promethea. V případě potřeby je lze doplnit i použitím aplikace Node-RED, která dokáže sbírat data z velkého množství různých zařízení. Zároveň je případně možné data předzpracovat přímo v Node-RED, což může být ve specifických aplikacích výhodné.

Další požadovaná vlastnost je automatické objevování zařízení v síti. Tuto vlastnost podporuje také samotný Prometheus. Je to velmi přínosné při monitorování

dynamických prostředí. Dále to je vhodné pro systémy, kde se předpokládá rozsáhlejší škálování, protože se vyhneme manuálnímu nastavování většího množství datových zdrojů. Tato vlastnost je v rozsáhlejších systémech prakticky nezbytná pro efektivní monitoring.

Pro vizualizaci dat je naprosto perfektní Grafana. Podle mě a podle zdrojů [15], [16] a [17] je Grafana jedním z nejlepších vizualizačních nástrojů pro time-series data. Výhodná je také snadná implementace Grafany společně s Prometheem, jelikož tuto kombinaci doporučují i samotní vývojáři daných softwarů a existuje k ní velmi dobrá dokumentace. Přidělení uživatelských práv a rolí je nejlepší řešit přímo v Grafaně, protože poskytuje rozsáhlé možnosti nastavení. Takto jsou popsány v dokumentaci Grafany, zdroji [18]. Každý uživatel, který se do Grafany přihlásí, má přidělenou roli, k níž se váží oprávnění. Jednotlivá oprávnění nám definují akce, které může uživatel v systému provádět. V Grafaně existují tři druhy oprávnění. Administrátorská oprávnění specifikují možnosti konfigurace nastavení a zdrojů celého serveru. Organizační oprávnění upravují přístupy k jednotlivým dashboardům, upozorněním, pluginům, týmům uživatelů a dalším organizačním strukturám. Uživatel může mít roli pouze pro zobrazení, roli editora nebo roli administrátora. Poslední skupina oprávnění rozhoduje o přístupu uživatelů do jednotlivých dashboardů a složek. Celkově jsou tato oprávnění dostačující i pro větší a složitější organizace díky verzatilitě jejich konfigurace.

Interaktivita frontendu je důležitý aspekt monitorovacího systému, protože nejen IT expert chce přistupovat k požadovaným informacím, a proto je vhodné umožnit uživatelům interagovat s jednotlivými panely. Ať se jedná přímo o ovládání či spouštění nějakých akcí z dashboardu, nebo o nastavování zobrazovaného časového intervalu. Grafana má tyto možnosti interakce na vysoké úrovni i pro koncového uživatele. Dají se doinstalovat i pluginy, které jsou velmi interaktivní. Například se jedná o plugin Worldmap panel, který může zobrazovat data podle geolokace, a uživatel si tak může prohlížet informace o místech, které potřebuje.

Pro generování upozornění s efektivním nastavením thresholdů je opět ideální Grafana. Její Alerting systém je podle zdroje [19] založený na vytváření dotazů a výrazů, které mohou kontrolovat více datových zdrojů nezávisle na lokaci uložených dat. Grafana má ve své dokumentaci detailně popsaný systém upozornění. Grafana dokáže v základní konfiguraci posílat upozornění pomocí několika různých kanálů, mimo jiné umí posílat e-maily, zprávy na Slack, Telegram, Discord nebo na Microsoft Teams. Thresholdy jsou nastavované pomocí porovnávání výrazů nebo

konkrétních příchozích dat. Grafana například umí aplikovat matematické výrazy na dotazy a výsledky použít k porovnání s jinou hodnotou. Další možnost je tzv. redukce, kde se z time-series dat vybírá buď největší, nejmenší, nebo průměrná hodnota, anebo se vypočítá součet všech hodnot, nebo lze spočítat počet datových prvků. Předposlední variantou je převzorkování dat, kde Grafana převádí časové řady na řady s konzistentním časovým intervalem. Umí data také jednoduše převzorkovat, a to jak podvzorkovat, tak i data doplnit na základě okolních hodnot, případně doplnit neznámou hodnotou NaN. Prometheus lze také využít k upozorňování, k tomu je ale potřeba další aplikace, Alertmanager, kdy Prometheus jí posílá upozornění a aplikace je dále zpracuje. Tato varianta není tak dobrá jako samotná integrace v Grafaně, kterou jsem proto pro tuto část vybral.

Pro analýzu dat je Grafana velmi vhodná, protože díky mnoha různým druhům vizualizačních panelů lze analyzovat velké množství dat poměrně snadno. Grafana má v základní konfiguraci mnoho různých vizualizačních nástrojů, od heatmap přes histogramy a koláčové grafy až po uzlové grafy a sledování trendů. Agregace dat je nejjednodušší přímo v Prometheovi díky jeho integrovanému dotazovacímu jazyku PromQL, který umožňuje efektivní zpracování a případnou analýzu (v kombinaci s Grafanou) i velkých datových objemů. PromQL je ideální díky rozsáhlým možnostem dotazování. Je postavený na jazyce Go, se kterým má tudíž společné některé vlastnosti. Jednotlivé dotazy se snadno postupně skládají do sebe a díky dobré dokumentaci se tímto způsobem dají vybírat i na první pohled nesouvisející data.

Predikce dat je velmi široké téma, u Grafany jsem našel možnost machine-learningu, ale ta je bohužel pouze součástí placené verze Grafana Cloud, takže jsem tuto možnost dále nezkoumal. Prometheus má funkci predict_linear, která je využitelná k jednoduché predikci díky implementaci lineární regrese, tedy proložení bodů v grafu přímkou. Není to nejlepší varianta, ale například k monitorování toho, že za dvě hodiny dojde místo na disku, je to vhodné. Pro pokročilejší predikce bych v případě potřeby implementoval pomocí Node-RED ještě další aplikaci na to přímo určenou. Node-RED je také vhodné použít pro automatizaci reakce na specifickou událost, což je další z požadovaných vlastností kvalitního monitorovacího systému. Teoreticky lze v Node-RED vytvořit tok, který bude reagovat na příchozí požadavek při dané události. Jednoduchý monitorovací systém by se obešel i bez Node-RED, Prometheus a Grafana na první pohled vypadají dostatečně, ale právě pro takovéto složitější interakce systému s nějakým jiným systémem je Node-RED perfektní.

Při použití funkčních uzlů lze provést i jednoduchou analýzu příchozích dat. Pro samotné vykonání požadované akce se dá použít další uzel, který může vykonávat velké množství různých akcí. Stejně jako pro automatizaci lze Node-RED využít i jako API pro integraci s externím softwarem. Díky široké podpoře vstupních i výstupních uzlů a případné vlastní implementaci pomocí standardních uzlů lze snadno komunikovat s externími službami. Grafana má zabezpečení frontendu, autentizaci a autorizaci uživatelů na velmi dobré úrovni. Pro komunikaci s ostatními částmi systému je třeba použít vhodné protokoly pro zachování bezpečnosti architektury. Logování historie změn v systému a dohledatelnost operací se týká hlavně Grafany a případně změn v Node-RED, protože koncový uživatel nepotřebuje a ideálně by neměl zasahovat do chodu Promethea. Grafana sama loguje uživatelské aktivity, jako je například přihlášení nebo změny dashboardů, podle nastavení logovací úrovně. Monitorování nákladů spojených s vlastním provozem je závislé na způsobu hostingu systému. Například pro monitoring nákladů při hostingu na Amazonu (AWS) existuje pro Promethea několik exportérů dat. Integrace dat z vlastních aplikací a hardwarových zařízení je velmi jednoduchá, podporované formáty dat se snadno dají sbírat Prometheem a nepodporované formáty dat se dají jednoduše zpracovávat v Node-RED a následně sbírat Prometheem.

Kromě zmiňované kombinace vybraných komponent jsem zvažoval použití dalších nástrojů. Mezi ně patřily Zabbix a ELK Stack. Zabbix jsem v navržené architektuře nepoužil kvůli několika vlastnostem, které má výsledná architektura na lepší úrovni. První z těchto vlastností je škálovatelnost, respektive výkon systému při velkém škálování. Ačkoliv lze podle zdroje [20] použít proxy Zabbix servery pro distribuci, tak podle zdrojů [21] a [22] je Prometheus vhodnější varianta pro velmi velké sítě, protože po zhruba 10 000 uzlech začíná Zabbix trpět pod tíhou své externí relační databáze a nedostatkem spolehlivé správy pro větší clustery. Prometheus tedy těží ze svých jednoduchých principů, a tudíž je pro rozsáhlé sítě jednoznačně vhodnější. Další z porovnávaných vlastností je konfigurovatelnost, kde se různé zdroje rozchází v tvrzeních o snadnosti konfigurace. Prometheus se snadno nakonfiguruje pro jednodušší základní systémy, a když je potřeba, tak umožňuje rozsáhlejší možnosti konfigurace, které se ale nerealizují tak snadno. Prometheus, s přihlédnutím k požadavkům na jeho funkce, nabízí jak jednoduché, tak komplexní možnosti konfigurace. Zabbix má konfigurování přívětivější, ale nenabízí tolik možností jako Prometheus. V oblasti monitoringu rozsáhlých systémů je podle mého názoru vhodnější disponovat rozsáhlejšími možnostmi konfigurace i za cenu toho, že konfigurování je

složitější a méně uživatelsky přívětivé, protože tato skutečnost by neměla být překážkou pro odborníka realizujícího rozsáhlejší systém. Prometheus má také oproti Zabbixu více podporovaných nástrojů třetích stran, knihoven a datových exportérů. ELK Stack jsem nepoužil, protože největší výhoda ELK Stacku oproti Prometheovi je podle zdrojů [23] a [24] ve zpracování logů, ale v oblasti monitoringu ve smyslu sběru metrik, což je primární použití při monitoringu IoT zařízení, je naopak výhodnější použít Prometheus. Pokud by nastala potřeba sběru a zpracování logů, tak lze použít Node-RED nebo je možné přidat do architektury snadno implementovatelný software určený pro sběr a zpracování logů, jako je například Grafana Loki. Ve svém případu užití nepředpokládám sběr většího množství logů, a proto použiji Node-RED pro ukázku jeho širokých možností využití.

4.2 Docker

Pro samotnou realizaci tohoto systému jsem si vybral Docker, což je virtualizační nástroj, který pracuje s jednotlivými komponenty systému jako s tzv. kontejnery. To má mnoho výhod, protože každou danou komponentu máme v odděleném kontejneru. Kontejner neobsahuje celý systém, ale jen danou komponentu, aplikaci, a pro ni důležité soubory. Tím mi odpadá nutnost běhu celého operačního systému, který bych musel mít na svém notebooku s Windows virtualizovaný. Další výhoda spočívá v jednoduché práci s rozsáhlou architekturou díky automatizovanému nasazování aplikací. Jednotlivé aplikace se díky odděleným prostředím snadněji spouštějí, konfigurují a testují. To velmi zjednodušuje vývoj a nasazování daného systému. Také nemají přímý vliv na ostatní aplikace, což je vhodné i z hlediska bezpečnosti, protože když se útočník dostane do jedné aplikace, tak automaticky nemá přístup k ostatním částem systému. Dockerfile je soubor, který specifikuje obraz aplikace, závislosti, konfiguraci a příkazy spuštěné po startu kontejneru.

4.3 Prometheus, Grafana a Node-RED – konfigurace

Dle mého názoru je Prometheus ideální volba nejen pro tento případ užití z důvodu efektivní práce s time-series daty a snadného automatického objevování zařízení v síti. Také lze snadno přijímat data z různých zdrojů, což je v kontextu monitorování složitějších systémů klíčové. Další výhoda Promethea je snadná škálovatelnost

pomocí federací více instancí Promethea. Bezpečnost lze snadno zajistit díky nativní podpoře HTTPS. S Grafanou jsem pracoval už při vypracovávání bakalářské práce, kde tato aplikace byla jednou ze součástí rozsáhlejší architektury. Nicméně jsem se na ni nemohl soustředit tak, jak bych chtěl. Při rešerši informací pro tuto práci jsem se rozhodl ji využít i v tomto případě, protože je podle mého názoru jedním z nejsilnějších nástrojů pro vizualizaci nejen time-series dat díky pokročilé uživatelské interaktivitě a zároveň velké konfigurovatelnosti pro použití v rozsáhlých systémech. Grafana má také, stejně jako Prometheus, velkou výhodu v široké podpoře různých datových zdrojů, takže při případném rozšíření již existující architektury lze do Grafany snadno přidat další datové zdroje.

Docker má samostatnou aplikaci pro Windows, která se jmenuje Docker Desktop a práce s ní je velmi uživatelsky přívětivá díky přehlednému UI. Pro prvotní zprovoznění základní architektury jsem použil image prom/prometheus. K jednoduché prvotní konfiguraci jsem si vytvořil soubor prometheus.yml, který vypadá následovně:

```
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'temperature_sensor'
    static_configs:
      - targets: ['sensor_simulator:9090']
```

Tato konfigurace znamená, že Prometheus každých 15 sekund sbírá data z cíle sensor_simulator na portu 9090. Tímto cílem je aplikace, která funguje tak, že simuluje senzor teploty pomocí Python aplikace, která spouští jednoduchý webový server na portu 9090. Tento server poskytuje náhodnou teplotu v rozsahu 15 až 30 stupňů Celsia na adrese /metrics. Pro takto jednoduchou aplikaci, u níž nepředpokládám jakékoliv reálné použití, protože její jediný účel je generování jednoduchých dat, jsem se rozhodl použít webový microframework Flask. Jeho výhoda spočívá ve velmi snadném vývoji aplikace a v jejím jednoduchém nasazení. Aplikace sensor_simulator.py vypadá následovně:

```
from flask import Flask
import random
app = Flask(__name__)

@app.route('/metrics')
def metrics():
    temperature = random.randint(20, 30)
    return f"temperature_sensor {temperature}\n"

if __name__ == '__main__':
    app.run(host='localhost', port=9090)
```

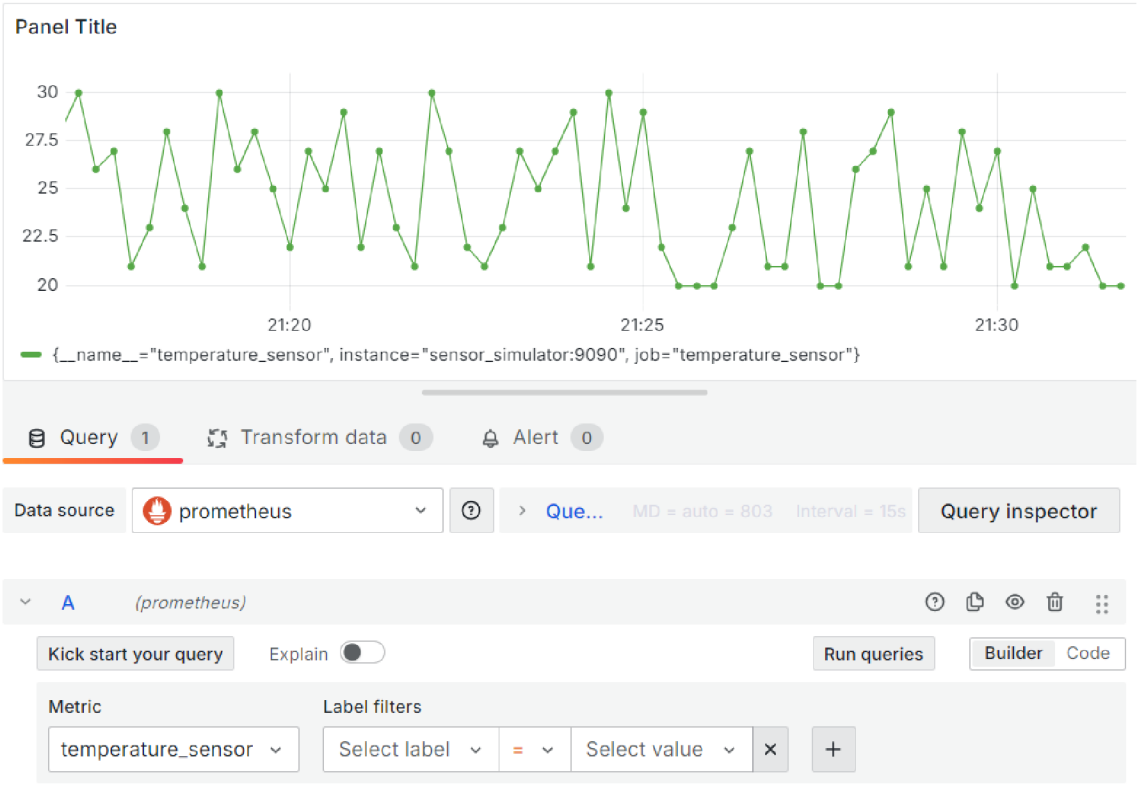
Prvotní test sběru dat a jejich vizualizace v Prometheovi je znázorněná na následujícím obrázku 4.1.

Obrázek 4.1: Prometheus – vizualizace dat

Tato aplikace samozřejmě není ideální, protože není nijak zabezpečená, ale pro otestování funkčnosti je dostačující. Dalšími kroky jsem tuto aplikaci zdokonalil. Nejdříve jsem přidal další části systému. Spustit Grafanu bylo ze součástí tohoto systému jednoznačně nejjednodušší, protože pro základní spuštění stačilo použít správný docker image a spustit kontejner, a to následovně:

```
docker run --net monitoring --name grafana -p 3000:3000 grafana/grafana
```

K zajištění základní bezpečnosti jsou komponenty systému připojeny do Docker sítě monitoring a v Grafaně je nutné při prvním přihlášení změnit heslo ze základní kombinace uživatelského jména „admin“ a hesla „admin“. Po přihlášení je přidání nového zdroje dat intuitivní. V sekci Connections - Data sources jsem vybral Prometheus jako zdroj dat a vzhledem k zatím chybějícímu zabezpečení jsem zadal jeho adresu http://prometheus:9090. V adrese je slovo „prometheus“, protože tak jsem pojmenoval Docker kontejner, ke kterému jsem Grafanu připojil. Po úspěšném připojení k Prometheovi jsem vytvořil jednoduchý panel, který vizualizuje time-series data v interaktivním grafu. Ve spodní části obrázku 4.2 je zvolená metrika temperature_sensor, která je generovaná v sensor_simulator.py, předávána do Promethea a následně přebírána a zobrazována Grafanou. Pro dashboard je v konfiguraci nastaven automatický interval obnovení, takže se dashboard automaticky aktualizuje s novými daty.



Obrázek 4.2: Grafana – přidání panelu

Co se týče integrace Node-RED do mnou navržené architektury, tak lze tuto integraci rozdělit na dvě nejprospěšnější varianty. První varianta znamená, že sbírám data z různých zařízení a ta data se skrze Promethea dostanou do Grafany. V Grafaně je následně případně zpracuji a vyhodnotím a při splnění určité podmínky budu chtít zareagovat nějak složitěji, než nabízí nativně Grafana, protože ta samotná zas tak rozsáhlé možnosti kromě upozornění nemá. V takovém případě je ideální variantou pro reagování na danou událost použít Node-RED. Druhá varianta využití Node-RED stojí úplně na počátku datového toku, a to v tom smyslu, že se nemusí z nějakého důvodu snadno posílat data do Promethea, a tak se použije Node-RED jako mezičlánek mezi monitorovaným zařízením a Prometheem. Mezi důvody pro použití takového mezičlánku může patřit například nějaké předfiltrování dat, přeformátovaní nebo jiné zpracování dat.

V Dockeru je opět samotné spuštění Node-RED velmi jednoduché díky existenci oficiálního image nodered/node-red, který stačí stáhnout a při spouštění kontejneru propojit mezi hostem a kontejnerem složku s konfigurací. To je důležité proto, aby se při opětovném spuštění kontejneru zachovala konfigurace. Propojení portu z kontejneru do Docker sítě „monitoring“ je samozřejmostí. Node-RED jsem zprovoznil k reagování na určitou událost. Tento proces začne v Grafaně, kde pro odeslání požadavku do Node-RED využiji webhook. Zdroj [25] popisuje webhook jako funkci zpětného volání založenou na protokolu HTTP. Je ideální pro snadnou komunikaci řízenou událostmi mezi dvěma aplikačními rozhraními, API. To je přesně to, co potřebuji, protože jedno API je Grafana odesílající požadavek do druhého API (Node-RED), kde se dále požadavek zpracuje. V nastavení Grafany jsem založil nový kontaktní bod, „contact point“, kde jsem jako způsob integrace vybral zmiňovaný webhook, nastavil HTTP metodu POST a jako URL zadal „http://nodered:1880/grafana-alert“. Následně jsem v Node-RED přidal první uzel http in. V něm jsem nastavil metodu POST a jako URL nastavil /grafana-alert. Tento uzel jsem propojil s debug uzlem, díky kterému lze zobrazit příchozí zprávy na předchozím uzlu. Po tomto nastavení Node-RED jsem v Grafaně provedl test kontaktního bodu a do Node-RED mi přišla testovací zpráva, která obsahuje informace o Grafanou vygenerovaném upozornění. Díky těmto informacím lze jednoduše identifikovat a reagovat na signalizovaný problém.

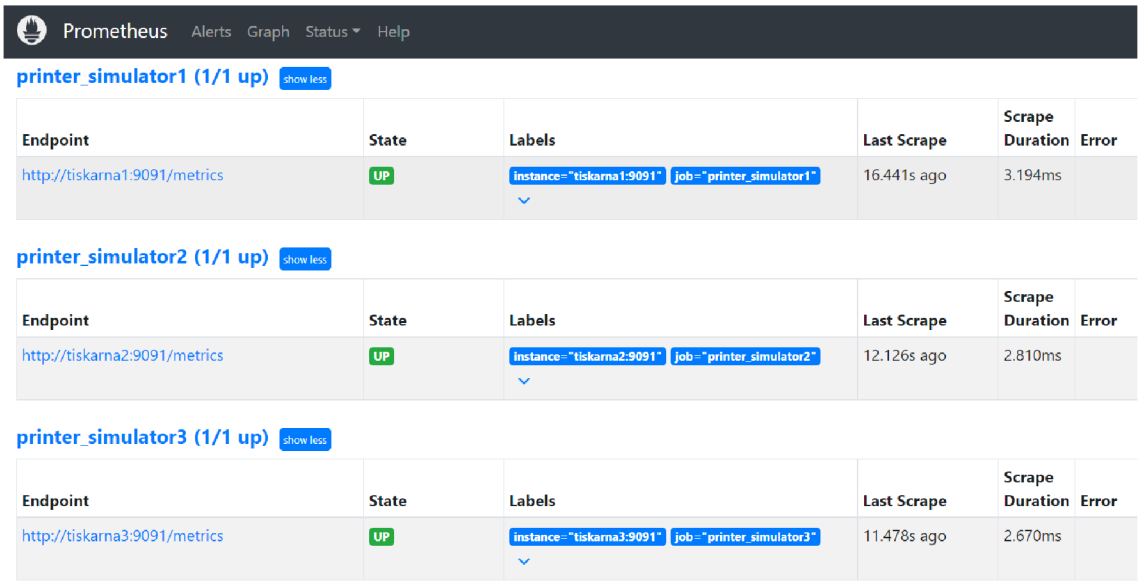
4.4 Rozšíření základní struktury

Ačkoliv nemám k dispozici velké výpočetní prostředky pro simulaci velmi velkého množství simulovaných zařízení, tak přesto architekturu rozšířím o další simulované zdroje dat a o funkce jednotlivých součástí systému. To je důležité pro demonstrování základních funkcí, rozhraní a vlastností vytvořeného systému.

Rozšíření základní struktury lze rozdělit do několika částí: první je rozšíření počtu simulovaných zdrojů dat. To byl první krok, protože bez něj nelze otestovat pokročilejší funkce systému. Další rozšíření jsou realizovány na straně Grafany a následně Node-RED, protože díky disponování více daty lze v Grafaně zpracovávat zajímavější vizualizace a s více metrikami lze implementovat komplexnější funkce. S tím souvisí i více možností pro posílání požadavků do Node-RED, v němž lze následně také provádět složitější operace a sestavit větší struktury uzlů a toků dat. Nad rámec této práce, například při reálném použití mnou navrhované architektury, by mohlo být pro některá využití vhodné implementovat externí úložiště pro dlouhodobé ukládání dat, protože Prometheus je stavěný jako krátkodobé datové úložiště. To z něj ale samozřejmě nedělá nevhodnou součást architektury, v teoretické části jsem popsal možnosti exportování dat do zvolené databáze. Pro předvedení požadovaných vlastností a funkcí to ale podle mého názoru není tak důležité, protože můj případ užití se zabývá monitoringem rychle se měnících procesů. Pro jiný případ užití s pomalejšími změnami procesu, a tudíž pro dlouhodobý monitoring by bylo vhodné takové úložiště implementovat. Další z možných rozšíření by byla automatizace nasazování požadovaných Docker kontejnerů a případná CI/CD pipeline pro automatizaci testování a nasazování nových verzí simulovaných aplikací. To bych provedl v případě správy rozsáhlejšího systému, což by bylo vhodné v případě potřeby simulace opravdu velké zátěže systému. Lépe by se poté daly demonstrovat výhody a nevýhody v rámci distribuovatelnosti a škálovatelnosti. Pro základní demonstrace vlastností ale takto velký systém není nutně potřeba, protože principy fungování architektury zůstávají stejné, případně velmi podobné.

Pro základní konfiguraci jsem vytvořil velmi jednoduchý skript generující náhodnou teplotu v nějakém rozsahu. To pro efektivní demonstraci požadovaných vlastností systému nestačí, a proto jsem se rozhodl v návaznosti na případ užití farmy 3D tiskáren vytvořit reálnější simulátory dat. První simulátor simuluje samotnou 3D tiskárnu. U 3D tisku jsou z mé zkušenosti nejdůležitější tyto informace: aktuální procento, dokončení tisku modelu, stav tiskárny (jestli tiskne, nebo netiskne) a tep-

loty trysky a vyhřívané podložky. Při vytváření aplikace printer_simulator.py, která simuluje průběhy zmiňovaných nejdůležitějších veličin, jsem použil ChatGPT 4 a již zmiňovaný webový microframework Flask. Aplikace simuluje bezchybný provoz 3D tiskárny, ve kterém se střídají fáze tisknutí a netisknutí, procenta tisku přibývají podle odhadu doby tisku a uplynulého času a teploty trysky a vyhřívané podložky v průběhu tisku kolísají. Po dokončení tisku se teploty začnou postupně snižovat až na teplotu okolí, která je v simulované tiskárně nastavená na 22 stupňů. Aplikace na HTTP endpointu poskytuje data ve formátu vhodném pro Prometheus. Dockerfile jsem použil téměř stejný jako u prvního testovacího sensor_simulator.py. Po nasazení první testovacího simulátoru tiskárny jsem využil Docker a spustil pět kontejnerů, kde každý z nich simuluje jednu tiskárnu. Na následujícím obrázku 4.3 je vidět korektní stav simulátorů tiskáren v prostředí Promethea.



Obrázek 4.3: Prometheus – webové prostředí

Jako další simulovaný zdroj dat by bylo vhodné mít držáky tiskové struny, které mají integrovanou váhu a odesílají periodicky informaci o aktuální zbývající hmotnosti materiálu. To by v reálné situaci bylo další zařízení, ale protože chci, aby simulace dat simulovala data alespoň přibližně realistická, tak jsem kód pro generování těchto dat přidal do aplikace printer_simulator.py. Zařadil jsem ho tam, protože pro realistické generování takových dat je nutné mít informaci o tom, zda tiskárna tiskne, nebo ne. Proto jsem dopsal následující kód, který odečítá hmotnost.

```
if cyklus == "tiskne":
 filament_hmotnost -= 2
 if filament_hmotnost <= 270:
     filament_hmotnost = 1260
     time.sleep(60)
```

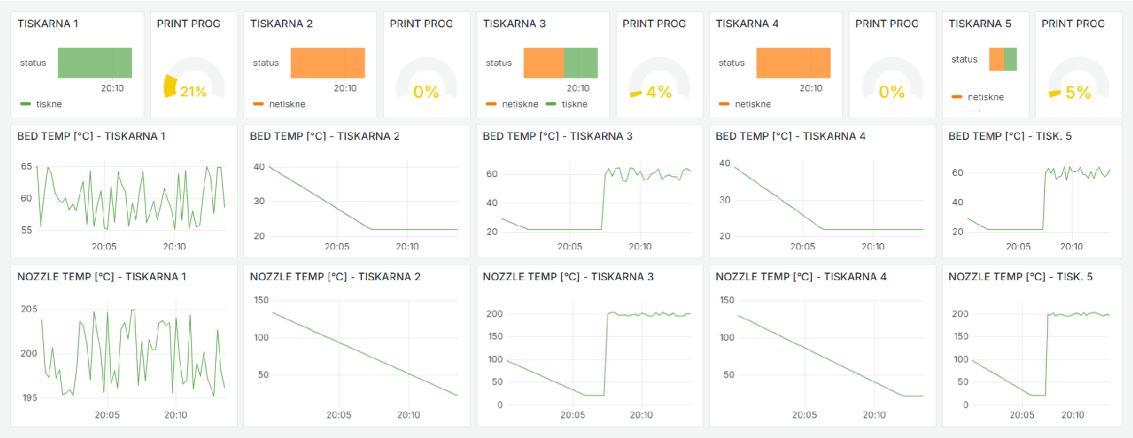
Za každých 15 sekund tisku se odečtou 2 gramy. To je hodnota vhodná pro ukázku vizualizace v Grafaně, protože typická 3D tiskárna tiskne řádově pomaleji. Při poklesu hmotnosti na 270 gramů se běh programu zastaví na minutu a čítač hmotnosti se resetuje na základních 1 260 gramů, což simuluje pauzu tisku a výměnu tiskové struny. Hodnota 270 gramů se odvíjí od předpokládané váhy samotné plastové špulky na strunu 260 g a zbývajících 10 gramů představuje zbytek tiskové struny, který se nedá již vytisknout kvůli typickému způsobu uchycení tiskové struny na špulce. Základních 1 260 gramů je odvozených od 1 000 gramů tiskové struny a již zmiňovaných 260 gramů samotné plastové špulky. V Grafaně jsem přidal panel pro zobrazení aktuální hmotnosti. K tomu jsem použil znovu panel typu Gauge, u kterého jsem nastavil threshold na 270 g, při nichž by měla nastat výměna tiskové struny. Pro ukázku využití Node-RED jsem v Grafaně přidal pravidlo, které odešle upozornění do Node-RED pokud klesne hmotnost pod určitou úroveň. Pokud by například byl v tiskové farmě nějaký automatický systém pro výměnu tiskových strun, tak by bylo vhodné ho následně ovládat z Node-RED. To je výhodné, protože se poté nemusí brát v potaz kompatibilita systému upozornění v Grafaně s tímto externím systémem, ale stačí správně nakonfigurovat výstupní uzel v Node-RED pro tento konkrétní systém.

4.5 Vizualizace v Grafaně

Grafana je velmi uživatelsky přívětivá a nastavování vizualizací dat je poměrně jednoduché. Grafana funguje na principu dashboardů a panelů. Dashboard je základ celého systému, je to vysoce konfigurovatelné interaktivní rozhraní Grafany. Do každého dashboardu lze přidat jednotlivé panely, které zobrazují různá data podle nastavení uživatelem. Dashboard má také různé interaktivní prvky, uživatel může měnit časový rozsah zobrazovaných dat, a to jak podle konkrétních časů, tak i podle výběru na časové ose přímo na panelu time-series vizualizace dat. Grafana také umožňuje na dashboardu konfigurovat proměnné, které mohou být použity pro

dynamické změny dotazů. Administrátor systému je schopný upravovat práva na zobrazení či editaci pro jednotlivé uživatele nebo skupiny.

Protože simuluji monitoring pěti simulovaných tiskáren, tak považuji za výhodné mít jeden dashboard, kde se zobrazují relevantní data ze všech tiskáren. Tento dashboard je zobrazen na obrázku 4.4.

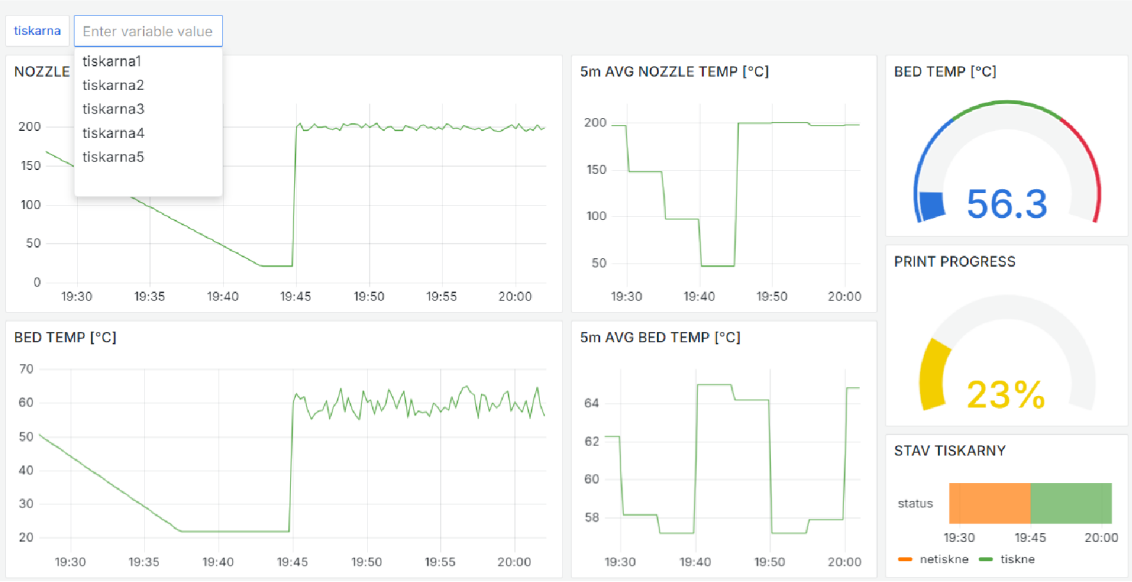


Obrázek 4.4: Grafana – vizualizace dat ze všech tiskáren

Pro každou tiskárnu jsem vytvořil čtyři vizualizační panely. Jeden panel ukazuje status tiskárny v průběhu času, což znamená, že vidím, kdy tiskárna tiskla a kdy netiskla. Protože v printer_status není jen samotný status. ale i další data, tak jsem musel použít v Grafaně nástroj Transform data, kterým jsem vyfiltroval jen potřebná data pro tento vizualizační panel. Potřebná data jsou čas a status. Ty jsem v grafickém rozhraní zaškrtl v dané záložce při editování panelu. Panel je typu state timeline, časová osa stavu. Pro rozlišení stavu na první pohled jsem v kartě value mappings, přiřazování hodnot, přidal podmínku, která při hodnotě „tiskne“ nastaví barvu vizualizace na zelenou a při hodnotě „netiskne“ na oranžovou. Při najetí myší na konkrétní stav tiskne/netiskne v panelu vidíme délku trvání toho konkrétního stavu. Další panel na dashboardu ukazuje aktuální procento dokončení tisku modelu. Tento panel je typu gauge, to znamená měřidlo, budík, díky kterému vidíme procentuální hodnotu a vizualizační půlkruh, který se s přibývajícími procenty více zaplňuje. Pokud tiskárna netiskne, tak hodnota zůstává na 0 %. Poslední dva panely jsou typu time-series a ukazují na časové ose průběh teploty v °C. Jeden z nich ukazuje teplotu podložky tiskárny, která je důležitá kvůli přilnavosti a kroucení modelu při tisku. Druhý zobrazuje teplotu trysky, která je důležitá pro kvalitní roztavení a následné vytlačení plastu. Tyto panely mají na ose x časovou osu, která se mění

automaticky podle zvoleného časového intervalu.

Výběr dat pro zobrazení správných dat ze správné tiskárny je v Grafaně velmi intuitivní. Při vytváření nebo editaci panelu se používají queries neboli dotazy. Ty slouží pro specifikaci dat, která chceme ze specifikovaného zdroje získat. Grafana podporuje hodně různých druhů zdrojů dat. V kombinaci s Prometheem jako zdrojem dat má Grafana k dispozici dotazovací jazyk PromQL. Dotazování pomocí PromQL má velmi dobře zpracovanou dokumentaci, ze zdroje [26] tedy vycházím při následujícím popisu. PromQL umožňuje vybírat a agregovat time-series data v reálném čase. Výsledná data následně Grafana zobrazuje podle nastavení dashboardu uživatelem. Co se týče datových typů, tak výrazy mohou pracovat s jedním ze čtyř datových typů - instantní vektor, vektor rozsahu, skalární hodnota nebo string. Instantní vektor je time-series soubor dat, který obsahuje jeden vzorek pro jednu časovou řadu. Vektor rozsahu má více datových bodů pro více časových vzorků a používá se tedy například při výpisu dat za určitý časový interval. To se v mém případě užití může hodit například pro výpočet průměrné teploty trysky nebo podložky tiskárny za určitou dobu, což může odhalit poruchu v řízení teploty nebo nějaký dlouhodobější výkyv. Vytvořil jsem proto nový dashboard v Grafaně, který slouží pro detailnější vizualizace pro zvolenou tiskárnu. Tento dashboard je zobrazen na obrázku 4.5.



Obrázek 4.5: Grafana – vizualizace dat z jedné tiskárny

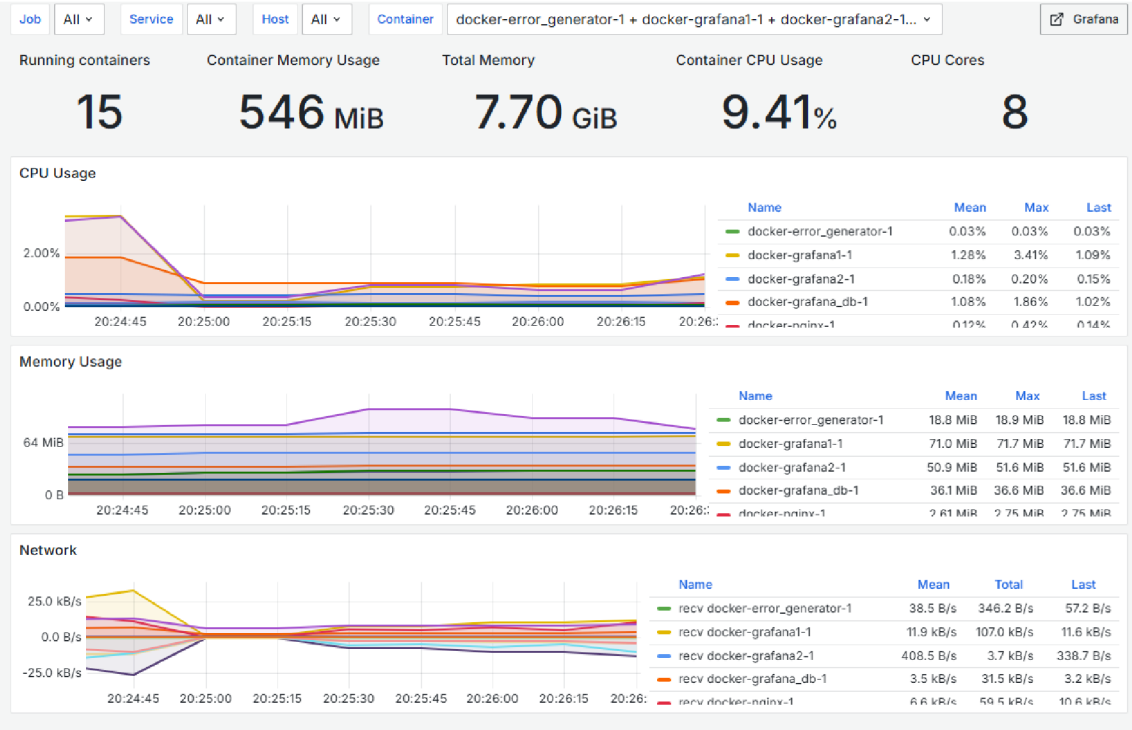
Pro zvolení požadované tiskárny jsem použil v dashboardu proměnnou, jejíž hodnotu uživatel vybírá v horní části dashboardu. Pomocí následujícího dotazu zobrazuji na panelu „5m AVG BED TEMP [°C]“ průměrnou teplotu v pětiminutových intervalech.

```
avg_over_time(printer_bed_teplota{printer_id="$tiskarna"}[5m:5m])
```

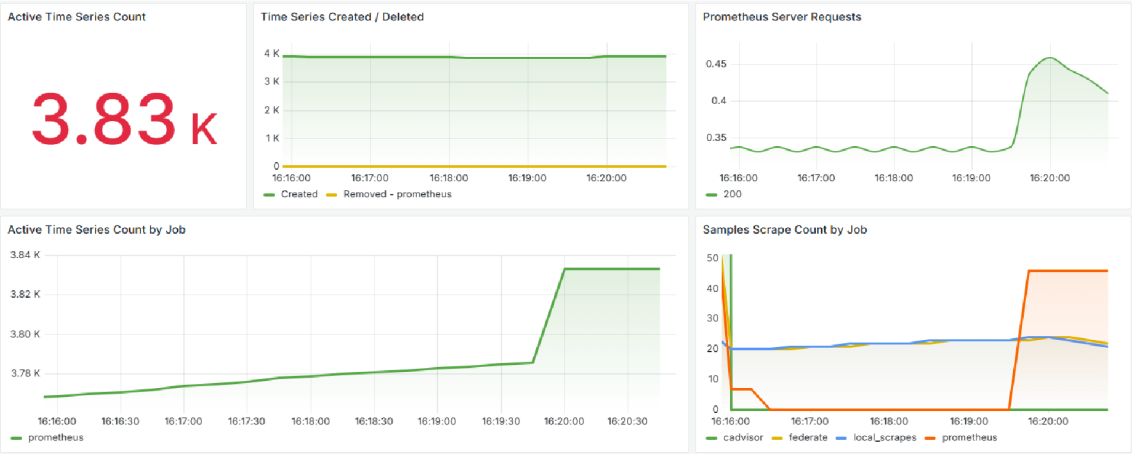
V tomto dotazu se pro každých 5 minut vybere a zprůměruje teplota podložky zvolené tiskárny. Dotazovací jazyk PromQL má intuitivní práci s časem díky jednoduché specifikaci časového intervalu. Například 5 minut je výše reprezentováno pomocí „5m“. Stejným způsobem se dají vybírat časové intervaly v řádu milisekund, sekund, minut, hodin, dnů, týdnů a let.

Poslední dva datové typy v dotazech PromQL jsou skalární hodnota a string. Skalární hodnota může být buď integer, nebo číslo s pohyblivou řádovou čárkou. String je textový řetězec, který je označený jednoduchými nebo dvojitými uvozovkami. Nebo také zpětným lomítkem v případě speciálních znaků, kde například `\n` označuje nový řádek. Pravidla pro tyto znaky přebírá PromQL z jazyka Go.

Grafana je vhodná nejen pro vizualizaci dat sbíraných z koncových senzorů, ale hodí se i pro vizualizace informací o jednotlivých komponentách samotného systému. V rámci mé implementace pro vizualizaci takového možného sběru dat sbírám data z Dockeru o jednotlivých kontejnerech, které reprezentují jednotlivé komponenty systému. Pro jednoduchý sběr těchto dat jsem přidal aplikaci cAdvisor, což je daemon od firmy Google, který snadno zpřístupňuje data o běžících kontejnerech Dockeru. Tato data následně zpřístupňuje pro sběr Prometheem, díky kterému se následně dostanou do Grafany, kde se vizualizují. Pro vizualizaci v Grafaně, jak je vidět na obrázku 4.6, jsem použil již existující šablonu (template) dashboardu „Docker monitoring“ ze zdroje [27]. Grafana umožňuje vizualizovat metriky výkonu Promethea, což je klíčové pro monitorování jeho zátěže. Pro vizualizaci těchto dat jsem použil existující dashboard ze zdroje [28]. Tento dashboard je na obrázku 4.7.



Obrázek 4.6: Grafana – vizualizace dat z Docker kontejnerů



Obrázek 4.7: Grafana – vizualizace metrik výkonu Promethea

4.6 Nasazování a vývoj

Při častějších úpravách kódu se v mé architektuře muselo po každé z nich provést několik kroků pro nasazení aplikace. Patřilo mezi ně smazání konkrétního kontejneru, sestavení nového Docker image a následné vytvoření nového kontejneru. To je

v případě 5 simulovaných tiskáren a s předpokladem rozšiřování architektury velmi nepraktické. Proto má Docker ideální řešení, které se jmenuje Docker Compose. Je to nástroj, díky němuž se snadno definují a pouští aplikace, které mají více kontejnerů. Odpadá tak nutnost manuálního nastavování a spouštění jednotlivých kontejnerů, protože se konfigurace provádí pomocí konfiguračního docker-compose.yml souboru. Po vytvoření takového konfiguračního souboru lze celou architekturu spustit pomocí jednoho jednoduchého příkazu. Další výhoda je možnost postupného spouštění na sobě závislých kontejnerů, například v mnou navrhované architektuře je vhodné spustit nejdříve Prometheus a až poté Grafanu kvůli závislosti Grafany na datech z Promethea. Následující kód je ukázka části souboru docker-compose.yml. Tato část spouští Node-RED.

```
node-red:
  image: node-red-with-docker
  user: root
  ports:
    - "1880:1880"
  networks:
    - monitoring
  volumes:
    - ./nodered:/data
    - /var/run/docker.sock:/var/run/docker.sock
```

4.7 Využití Node-RED

V předchozích kapitolách jsem popsal několik konkrétních využití aplikace Node-RED v rámci mnou navržené architektury. Obecně mohu rozdělit relevantní uzly do třech kategorií: uzly vstupní, výstupní a uzly nějakým způsobem zpracovávající data mezi vstupem a výstupem z Node-RED. Jako vstupní uzel je vhodné použít „http in“ uzel, který se snadno propojuje se systémem upozornění v Grafaně. To je snadné díky podpoře webhooků v Grafaně. Další možný vstupní uzel by byl uzel WebSocket, který umožňuje pomocí TCP (zkratka z anglického „Transmission Control Protocol“, česky protokol řízení přenosu) spojení komunikovat s jinými aplikacemi. Grafana ale toto nepodporuje, proto jsem v ukázkovém řešení ponechal webhook a uzel http-in. V Grafaně jsem nastavil pravidlo pro odeslání informace

do Node-RED pomocí webhooku o dosažení 95 % dokončení tiskové úlohy. V Node-RED jsem ve Flow 3 použil uzel http-in pro příjem informace, http response pro odpověď a uzel debug pro zobrazení informace. Po vhodném rozšíření systému by bylo možné v Node-RED reagovat na danou událost například rozsvícením světel u konkrétní tiskárny. Node-RED také podporuje MQTT (zkratka z anglického „MQ Telemetry Transport“) vstupní a výstupní uzly. To už je pro můj případ užití relevantní, protože při velkém počtu zařízení může být výhodné použít více Node-RED instancí. Pokud by se nerozdělil tok dat do různých instancí Node-RED přímo v Grafaně, tak lze tento systém řídit pomocí MQTT brokera. To může mít tu výhodu, že se sníží zátěž sítě nebo přímo výpočetního serveru na straně Node-RED, který při použití MQTT brokera nemusí sám složitějším způsobem routovat příchozí požadavky. MQTT je podle zdroje [29] vhodné pro situace s malou šířkou pásma sítě nebo s velkou latencí a podle zdroje [30] je MQTT vhodné pro použití v IoT aplikacích.

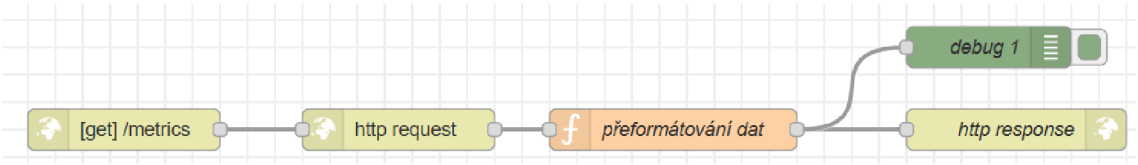
Pro další ukázku, jak jinak lze využít Node-RED, jsem se rozhodl začlenit strukturu pro implementaci dat, která nejsou v ideálním formátu pro další zpracování Prometheem a následně Grafanou. Důvodem tohoto rozhodnutí bylo, že se taková situace může v monitorovaném systému stát velmi snadno, protože ne u všech zařízení, které chceme použít, si část pro export a odesílání dat píše nebo definuje člověk sám. Díky využití Node-RED se data snadno zpracují a převedou do formátu, který lépe vyhovuje dalším částem architektury kvůli následným operacím.

V rámci svého případu užití jsem se rozhodl vytvořit simulátor výstupu z nějakého nativního softwaru tiskárny, který vytváří chybová hlášení. Ta jsou ale ve formátu, který není strukturovaný pro další použití, a proto je potřeba použít Node-RED. Pro generování chybových hlášení jsem použil podobnou strukturu, jako má printer_simulator.py. Chybová hlášení generuje Python aplikace error_generator.py, která používá jednoduchý webový microframework Flask a pomocí jednoduché logiky vybírá chyby z listu a přidává k nim čas a datum. Tato aplikace generuje data v následujícím formátu:

```
{"log":"Date: 2024-04-18 Time: 21:47:27 Error 404: Filament not found"}
```

Takový formát není žádaný, a proto jsem následně použil Node-RED. Tok dat v Node-RED začíná uzlem http in, který má nastavenou metodu GET, protože Prometheus data získává z endpointů právě pomocí této metody. Po přijetí požadavku se aktivuje další uzel http request, který má za úkol sběr dat z endpointu error generá-

toru. To se provádí také metodou GET, ale daný endpoint vrací data v nevhodném formátu. Proto následuje další uzel, kterým je uzel function, „přeformátování dat“. V tomto uzlu je krátký JavaScript kód, který má jako vstup zprávu z předchozího uzlu, nevhodně naformátovaná data, a jako výstup data naformátovaná vhodně. Toho jsem docílil pomocí jednoduché analýzy (parsování) nevhodně naformátovaných dat, protože předpokládám pořád stejně formátovaný vstup. To samozřejmě nelze tvrdit obecně, ale pro reálné použití lze přidat delší a komplexnější skript pro analýzu a následné formátování příchozích dat. Celý tok dat končí uzlem http response, který vrací odpověď se správně naformátovanými daty. Tok je vidět na následujícím obrázku 4.8.



Obrázek 4.8: Node-RED – Tok 1

V Grafaně jsem přidal panel pro zobrazení příchozích dat ve správném formátu, který je vidět na následujícím obrázku 4.9.

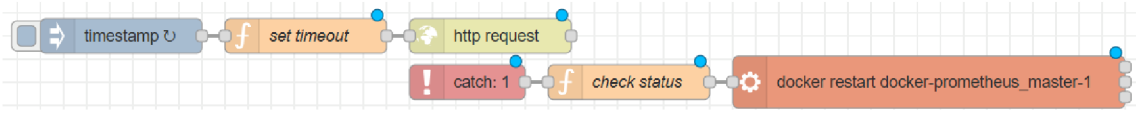
Time	__name__	date	error_code	error_message	instance
2024-04-20 15:40:30	error_nodered	2024-04-20	555	Motherboard overheat	node-red:1880
2024-04-20 15:40:45	error_nodered	2024-04-20	555	Motherboard overheat	node-red:1880
2024-04-20 15:41:00	error_nodered	2024-04-20	555	Motherboard overheat	node-red:1880
2024-04-20 15:41:15	error_nodered	2024-04-20	555	Motherboard overheat	node-red:1880
2024-04-20 15:31:00	error_nodered	2024-04-20	567	Motor overload	node-red:1880
2024-04-20 15:31:15	error_nodered	2024-04-20	567	Motor overload	node-red:1880
2024-04-20 15:31:30	error_nodered	2024-04-20	567	Motor overload	node-red:1880

Obrázek 4.9: Grafana – vizualizace příchozích dat ve formě tabulky

Pokud by bylo potřeba zpracovávat větší objem logů, tak by bylo vhodné do architektury přidat komponentu určenou pro zpracování logů. Tou by mohla být například aplikace Grafana Loki. Ta je definována zdrojem [31] jako „Horizontálně škálovatelný, vysoce dostupný systém agregace logů pro více uživatelů inspirovaný systémem Prometheus. Loki se od systému Prometheus liší tím, že se zaměřuje na logy namísto metrik a shromažďuje logy pomocí push, nikoli pull.“ Výhodou tohoto

řešení je snadná integrace s Grafanou. Jelikož v rámci mého případu užití není nutné sbírat logy, tak tato aplikace nebyla použita. Sběr logů pomocí Node-RED především ilustruje jeho široké možnosti využití.

Node-RED jsem se také rozhodl použít pro ukázku pokusu o obnovení funkčnosti systému v případě výpadku nějaké komponenty. Vybral jsem si pro to ukázkový příklad, kdy z nějakého důvodu přestane fungovat Prometheus. Pro kontrolu a následný pokus o řešení této situace jsem vytvořil další tok dat v Node-RED. Ten začíná uzlem inject, který s definovaným intervalem generuje zprávu, která inicializuje akci v dalším uzlu. Druhým uzlem je uzel function, který nastavuje časovou prodlevu následujícího uzlu. Následujícím uzlem je uzel http request, který pomocí protokolu HTTP a jeho metody GET kontroluje stav Prometheus serveru, který je dostupný na jeho endpointu /healthy. Ten vrací status code, díky kterému lze určit stav dotazovaného serveru. Proto dalším uzlem v datovém toku je uzel function, v němž se kontroluje právě onen status code. Pokud to není status 200, což znamená OK, tak v toku dál postupuje informace o nutnosti restartu. Pokud to je status 200, tak se žádná další informace nezasílá. Další uzel je tedy aktivován pouze v případě neočekávaného chování serveru Prometheus. V mé ukázkové implementaci je tento uzel typu exec, protože spouští kód, který restartuje docker kontejner, ve kterém je Prometheus spuštěný. Aby bylo možné ovládat Docker z kontejneru Node-RED, tak jsem vytvořil vlastní Docker image, který je založený na oficiálním image Node-RED a byla do něj doinstalovaná komponenta docker-cli kvůli komunikaci s daemonem. Další nutná úprava byla přidání Unix socketu pro komunikaci s Dockerem. To bylo realizováno pomocí jednoduchého prolinkování souboru docker.sock do kontejneru None-RED. Otestoval jsem funkčnost celého toku pomocí jednoduchého pozastavení běhu kontejneru s Prometheem a aplikace Node-RED správně vyhodnotila nefunkčnost a kontejner se restartoval podle očekávání. Samozřejmě nelze všechny problémy řešit pouhým restartováním kontejneru, ale tok v Node-RED lze snadno doplnit o logiku, podle které se může řešit nefunkční komponenta jinými způsoby. Tok je vidět na následujícím obrázku 4.10.



Obrázek 4.10: Node-RED – Tok 2

4.8 Bezpečnost navrhovaného systému

Mnou navržená architektura a realizovaná ukázková aplikace se jeví jako méně bezpečná z důvodu jednoduché komunikace mezi jednotlivými komponentami. Ta totiž probíhá probíhá bez šifrování, pouze přes protokol HTTP. Pro zvýšení bezpečnosti jsem se rozhodl implementovat místo protokolu HTTP protokol HTTPS kvůli poměrně jednoduchému implementování a zároveň velké úrovni zabezpečení. Vzhledem k mnou navrhované architektuře je vhodné použít reverzní proxy server a komunikaci mezi samotnými komponentami zabezpečit pomocí konfigurace, která pro komunikaci používá HTTPS protokol. Reverzní proxy server Nginx zajišťuje zabezpečení komunikace mezi uživatelem a komponentami architektury. Pro jeho použití jsem se rozhodl nejen kvůli zmiňované bezpečnosti, ale také kvůli možnosti využití pro load balancing, což je rozdělování požadavků mezi více instancí komponenty, čímž dochází k optimalizaci využití zdrojů. To je v architektuře zaměřené na distribuovatelnost a škálovatelnost velmi výhodné, v předchozích částech této práce jsme zmiňoval se stejnou motivací MQTT broker.

Jako reverzní proxy server jsem se rozhodl použít Nginx. Ten jsem vybral kvůli vysoké efektivitě, co se týče systémových zdrojů, kvůli podpoře komplexních konfigurací, kvůli jednoduchému použití jako reverzního proxy serveru a také kvůli velké rozšířenosti a z toho plynoucího velkého množství zdrojů a dokumentace. Pro zprovoznění Nginxu jsem postupoval následovně. Stejně jako ostatní komponenty aplikace je ze stejných důvodů žádoucí instalace Nginx jako Docker kontejneru. Proto jsem z oficiálního repozitáře Dockeru stáhnul image nginx. Do již existujícího docker-compose jsem následně přidal konfiguraci pro Nginx. Ta se sestává ze zrcadlení portu a předávání souboru s konfigurací a složky s certifikáty. Závislost Nginxu jsem nastavil na ostatních aplikacích kromě těch, které generují data. Nginx je také ve stejné Docker síti jako ostatní součásti systému.

Pro zabezpečení pomocí HTTPS je potřeba vygenerovat SSL certifikát, díky kterému je zaručena identifikace a šifrování komunikace pomocí asymetrické kryptografie. Pro vygenerování certifikátu je jedna z nejjednodušších variant použití openssl. Pomocí něj jsem na svém lokálním Ubuntu následujícím způsobem vygeneroval svůj soukromý klíč a veřejný certifikát, který následně bude potřeba pro správné nastavení HTTPS v Nginxu.

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048
-keyout ./nginx.key -out ./nginx.crt
```

Tento příkaz mi vygeneroval soukromý RSA klíč dlouhý 2 048 bitů a samopodepsaný certifikát. Samopodepsaný certifikát znamená, že certifikát nebyl vystavený a podepsaný certifikační autoritou, ale uživatelem nebo organizací, která certifikát používá. Z toho vyplývá, že takový certifikát není implicitně důvěryhodný pro jiné uživatele a servery. To ale není problém v případě mé izolované testovací sítě. Pokud bychom nasazovali mnou navrhovanou architekturu na reálné řešení v síti, kde je nebezpečí napadení větší, tak by bylo vhodné použít pro vygenerování požadovaného certifikátu uznávanou certifikační autoritu. Klíč jsem si uložil do souboru nginx.key a certifikát do nginx.crt. Tyto soubory jsem poté přesunul do složky certs, kterou jsem sdílel dovnitř Docker kontejneru Nginxu.

Konfigurace samotného reverzního proxy serveru je jednoduchá, všechna nastavení jsou v souboru nginx.conf. V něm jsem specifikoval cesty ke klíči a k certifikátu, port a server_name, který je v mé konfiguraci localhost. Poté jsem nakonfiguroval samotné cesty k jednotlivým aplikacím. Ty jsem definoval v blocích location, kde jsem podle aplikace, na kterou chci přistupovat, respektive adresy, přiřadil cílovou adresu. Část souboru nginx.conf pro konfiguraci přesměrování pro Node-RED vypadá následovně.

```
location /node-red/ {
        proxy_pass http://node-red:1880/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
```

Tyto bloky jsem definoval pro každou aplikaci v mnou navrhované architektuře kromě Promethea, protože ten lze snadněji nastavit přímo v jeho konfiguraci. Stačí mu totiž předat soubor web-config.yml, klíč s certifikátem a v docker-compose nastavit při spouštění kontejneru s Prometheem příkazy, které správně nakonfigurují cesty k souborům, url a port, na kterém Prometheus poslouchá.

Pro zabezpečení sběru dat z koncových zařízení podporuje Prometheus autorizační hlavičku, kterou obsahuje každý požadavek na sběr dat (scrape request). Také lze implementovat OAuth 2.0, což je autorizační nástroj založený na tokenech.

4.9 Rozšiřitelnost

Rozšiřitelnost architektury je velmi důležitá vlastnost pro dlouhodobou relevantnost daného systému. Jedná se o umožnění přidávání nebo modifikace částí systému bez nějakého narušení stávající architektury. Pro monitorovací systémy je to potřebná vlastnost kvůli možným změnám monitorovaných zařízení, přidávání nových zdrojů dat nebo přidávání dalších částí systému pro zpracování monitorovaných zařízení. Pro demonstraci snadné rozšiřitelnosti je celá architektura zprovozněná v Dockeru, který řeší izolované prostředí pro každou část systému. Z toho vyplývá, že přidáním další aplikace se neovlivní ostatní části systému. Další výhoda je možnost nastavení kontejneru pro každou aplikaci zvlášť, a tudíž absence potřeby globálních změn, například co se týká env proměnných (proměnných prostředí). Pro snadné přidávání dalších aplikací je také ideální použití Docker Compose, který umožňuje snadnou konfiguraci všech spouštěných kontejnerů. Pro přidání nové aplikace stačí v konfiguračním souboru Docker Compose přidat jednoduchou definici, která obsahuje nastavení obrazu, portů a síťových nastavení a závislostí. Nová aplikace tedy nemá šanci změnit fungování ostatních služeb. Při rozšiřování svého systému jsem na praktické rozšíření stávající architektury narazil během zabezpečování komunikace mezi komponentami. Tím rozšířením bylo přidání Nginx reverzního proxy serveru. To se díky dobře navržené architektuře dělalo velmi snadno. Daný proces implementace této komponenty je popsán v předchozí části o bezpečnosti navrhovaného systému.

Rozšiřitelnost na úrovni přidávání dalších zdrojů dat je v mnou navržené architektuře také dobře realizovatelná. Prometheus je velmi univerzální co se týče sběru dat z různých datových zdrojů. Toho dosahuje pomocí exportérů, což jsou nástroje, které umožňují sběr dat v různých formátech za pomoci různých protokolů. Tyto exportéry specializované na sběr dat jsou dostupné z konkrétních databází nebo z různých hardwarů. Kdokoliv si může poměrně snadno vytvořit další exportér, protože pravidla a pokyny pro vývoj nových exportérů jsou dobře popsány v dokumentaci Promethea. Pokud by z nějakého důvodu nebylo možné nebo vhodné použít tento způsob, tak lze pro přidání dalšího zdroje dat využít aplikaci Node-RED. Ta má také škálu různých datových vstupů a výstupů. Její další výhodou je možnost jednoduššího zpracování dat před předáním do Promethea díky pokročilejším uzlům, které mohou být využity pro filtrování nebo pro pokročilejší operace s příchozími daty.

Rozšiřitelná je i vizualizační část architektury, kde Grafana nabízí mnoho mož-

ností pro rozšíření svých funkcí. Mezi ně patří například pluginy, kterých Grafana podporuje širokou škálu. Grafana poskytuje oficiální repozitář s již vytvořenými pluginy nebo umožňuje uživatelům si vytvořit vlastní. Pluginy mohou přidávat nové typy vizualizačních panelů, nové datové zdroje a další komplexnější aplikace provázané s Grafanou.

Mnou navržená architektura tudíž má, díky použití Dockeru, Node-RED a Promethea, celkovou rozšiřitelnost na velmi dobré úrovni. Je vhodná pro velký počet různých datových zdrojů a disponuje možnostmi pro pokročilé vkládání dat do relevantních částí systému. I samotná vizualizace dat je snadno rozšiřitelná díky použití Grafany.

4.10 Distribuovatelnost

Distribuovatelnost je pro systémy, které počítají s větším počtem monitorovaných zařízení, zásadní. Distribuovaný systém je takový systém, kde jednotlivé komponenty daného systému jsou provozovány odděleně a jsou propojeny pomocí sítě. To má za výhodu to, že celý systém není spuštěný na jednom fyzickém stroji, a tudíž lze snadno škálovat přidáváním dalších serverů. Díky oddělení jednotlivých komponent se také snáze modifikují. Pro zpracovávání větších objemů dat z většího množství zařízení je takový systém ideální díky rozdělení úloh mezi více procesorů.

Mnou navržená architektura je velmi snadno distribuovatelná. Konkrétně používám pro oddělení jednotlivých komponent Docker. Ten zaručuje to, že každá z komponent běží ve svém vlastním kontejneru, které simulují použití dalších výpočetních serverů. Některé výhody distribuovaných systémů, jako je například snadná modifikace jednotlivých aplikací, jsou zachovány i v prostředí Dockeru běžícím na jednom fyzickém stroji.

Další praktickou výhodou použití Dockeru, respektive distribuovaného systému, je možnost snadného restartu celé aplikace v případě nějakého problému. V této architektuře by se dal elegantně kontrolovat běh jednotlivých komponent pomocí Docker Health Check, který automaticky testuje běh aplikace. Dělá se to tak, že se v základním nastavení každých 30 sekund spustí předem definovaný skript, který ověřuje stav kontejneru. Po definovaném počtu neúspěšných pokusů může Docker automaticky restartovat kontejner a tím se pokusit ho vrátit do provozuschopného stavu.

4.11 Škálovatelnost

Škálovatelnost je jedna z dalších důležitých vlastností monitorovacích systémů, protože při větším počtu zařízení mohou být sběr, zpracování i vizualizace náročné na systémové prostředky hostitelského stroje. Díky možnostem škálování lze efektivně zvyšovat náročnost architektury bez poklesu výkonu. Toho se dá docílit buď horizontálním, nebo vertikálním škálováním.

Vertikální škálování znamená to, že se jednoduše zvýší výpočetní kapacita konkrétního kontejneru nebo stroje. Tato vlastnost velmi úzce souvisí s distribuovatelností zmíněnou dříve, protože když se zvýší potřeba výkonu u konkrétní aplikace, tak lze efektivně zlepšit výkonnost u konkrétního výpočetního uzlu. To může znamenat například přidání více paměti RAM, výměnu procesoru za lepší nebo zvětšení úložiště. Mnou navržená architektura je na vertikální škálování vhodná, protože všechny komponenty mohou běžet na jiném výpočetním uzlu, a tudíž se snadno identifikují uzly, které jsou přetěžované. Navíc lze systémové zdroje monitorovat přímo v systému, každý stroj může sdílet data o zatížení do Promethea a následně lze v Grafaně vytvořit vizualizační panel a případné upozornění o přetížení systému. Na to lze navázat akce v Node-RED, například automatické zvětšení alokované paměti pro danou komponentu. U všech těchto variant se ale dostaneme na limity ve větších systémech velmi snadno, a proto se v podobných systémech klade důraz hlavně na škálovatelnost horizontální.

Horizontální škálování velmi úzce souvisí s distribuovatelností, protože případná potřeba většího výkonu systému se řeší pomocí přidávání dalších výpočetních uzlů. Na to je mnou navržená architektura perfektně připravená, protože jednotlivé části systému jsou od sebe oddělené, a tudíž se mohou snadno přidávat prakticky bez zásahu do existujícího systému. V prostředí Dockeru stačí jen přidat kontejner a nastavit komunikaci s již existujícími kontejnery.

Pro systematické horizontální škálování Promethea je vhodné zavést tzv. federaci. To znamená, že se zavede stromová struktura více instancí Promethea. V ukázkovém případě jsem zavedl pouze dvě úrovně, master a slave, ale v rozsáhlejším systému jich může být více. Pro ukázku jsem použil dvě slave instance Promethea a nad nimi jednu master instanci. Ta je v mé ukázkové aplikaci nastavená tak, že přebírá všechny metriky ze slave instancí. To lze v rozsáhlejším systému nastavit i tak, že se budou do vyšších instancí sdílet agregovaná data a případné podrob-

nější informace v případě potřeby brát z instancí nižších. Například se to dá využít pro snížení počtu dat, u mého případu užití by se mohlo jednat například o sdílení průměrné teploty trysky tiskárny za posledních pět minut, čímž lze značně snížit zatížení jak samotných vyšších instancí, tak samotné sítě. V Grafaně není poznat implementace této struktury, protože jako zdroj dat nastavuji master Prometheus, který má všechna data z instancí pod sebou. Pro ověření funkčnosti jsem k jedné slave instanci přiřadil tři simulátory tiskáren a ke druhé slave instanci dva simulátory. Slave instance se nastavují stejně jako standardní Prometheus instance, rozdíl je u master instance, u které je potřeba definovat, jaká data se mají sbírat ze slave instancí a které instance to jsou. Na tomto příkladě lze demonstrovat, jak snadno se škáluje tato část mnou navrhované architektury.

Podobně jako lze horizontálně škálovat Prometheus, tak lze škálovat i Grafanu. Grafana je sice poměrně nenáročná na systémové prostředky, ale v rozsáhlejším systému, který například může využívat více lidí nebo budou existovat větší nároky na výkon Grafany z jiného důvodu, nemusí být ideální používat jednu instanci tak, jako to bylo v mnou navržené architektuře na počátku. Proto jsem se rozhodl implementovat ukázkové řešení škálování Grafany. Přímo v dokumentaci Grafany je pro vysokou dostupnost docílenou více instancemi doporučeno použít externí databázi pro ukládání dashboardů, panelů, uživatelů a dalších dat. Jako tuto databázi je vhodné použít Postgres, který jsem přidal do docker-compose. Pro použití více instancí Grafany je zásadní load balancing server. Nginx je vhodná varianta pro toto použití a zároveň jsem ho použil jako reverzní proxy server. Load balancing server je takový server, který umožňuje rozložení zátěže na více koncových instancí aplikace. V konfiguračním souboru Nginxu jsem nastavil upstream, který definuje cluster cílových serverů, na dvě ukázkové instance Grafany. Tyto dvě instance jsou definované v docker-compose a oproti standartní konfiguraci Grafany se téměř neliší. Pouze mají definované stejné environment proměnné pro přihlašování do databáze, jejíž adresa je také definovaná jako environment proměnná. Protože se databáze spouští déle a je nutné, aby se instance Grafany spouštěly až po spuštění databáze, tak je do docker-compose zavedená logika, která kontroluje běh databáze. Ta funguje na principu tzv. healthchecku, pomocí kterého se kontroluje stav kontejneru. Část kódu v docker-compose vypadá následovně.

```
healthcheck:
    test: ["CMD", "pg_isready", "-U", "user"]
    interval: 30s
    timeout: 30s
    retries: 5
```

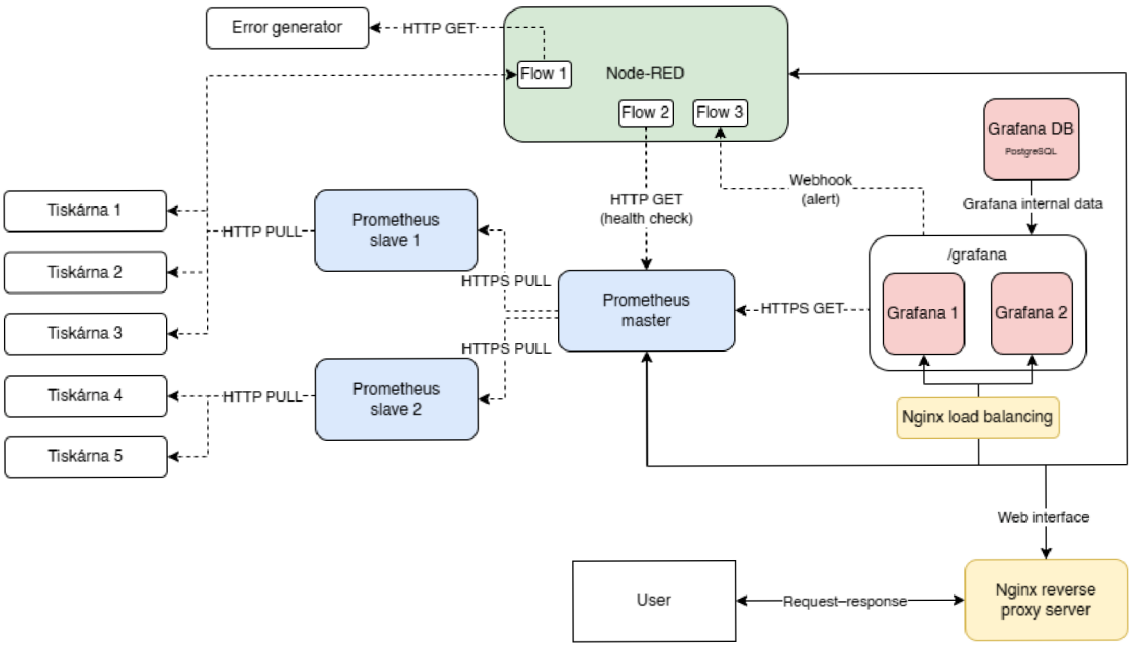
Healthcheck se tedy provádí tak, že se v kontejneru spustí příkaz pg_isready a na základě jeho vráceného stavu je healthcheck vyhodnocený jako úspěšný, nebo ne. Příkaz pg_isready je nástroj PostgreSQL, díky kterému lze ověřit to, zda je server PostgreSQL schopný přijímat požadavky. Pokud je PostgreSQL správně spuštěný, tak je healthcheck vyhodnocený jako úspěšný, a v důsledku toho se začnou spouštět kontejnery Grafany. V úryvku kódu lze také vidět nastavený interval kontroly, čas pro timeout a počet opakovaných pokusů. Tyto konkrétní údaje je vhodné případně změnit v případě použití ve větším systému, kde by se mohla databáze spouštět delší dobu. Řešení škálovatelnosti Grafany tedy není složité a nevyžaduje složitou konfiguraci.

Škálování Node-RED je vhodné pojmout méně obecně, respektive ke škálování Node-RED lze přistupovat dvěma způsoby. První způsob je, stejně jako u Grafany, založený na použití load-balanceru. To může být zvláště vhodné pro ty aplikace Node-RED, které nevyžadují koordinaci dat mezi jednotlivými instancemi. I taková potřeba se ale dá vyřešit například pomocí předáváním si MQTT zpráv mezi jednotlivými instancemi. Alternativa k MQTT je použití Redis, což je velmi rychlá databáze podporovaná systémem Node-RED. Uchovávat informace o stavech může Redis anebo lze použít pub/sub model, který funguje obdobně jako MQTT. V Node-RED lze použít balíček node-red-contrib-redis, který přidává uzly pro práci s Redis. Druhý způsob přístupu ke škálování Node-RED spočívá v rozdělení logiky, a tím pádem i systémových nároků na jednotlivé instance.

4.12 Finální podoba architektury

Motivací pro monitorovací systémy je sběr dat a jejich následné zpracování. Tato architektura je založená na sběru dat pomocí federace více instancí Promethea. Ta umožňuje snadno škálovatelný a distribuovatelný sběr dat z velkého počtu různých zařízení. Pro následné zpracování a vizualizaci dat je implementovaná Grafana, kterou lze také snadno distribuovat a škálovat díky navrženému řešení s použitou databází pro interní data Grafany a load-balancing serveru Nginx. Grafana je hlavním výstupem systému pro koncového uživatele, který snadno a interaktivně pracuje se sbíranými daty ve smyslu jejich vizualizace a analýzy. Velkou roli v architektuře také hraje aplikace Node-RED, která je použitá ve více různých scénářích od formátování vstupních dat přes kontrolu stavu jednotlivých komponent až po reakci na události vycházející z Grafany a jejího upozorňovacího systému.

Pro sběr dat existují v mnou navrhované architektuře dvě základní varianty. První varianta, ta ideální, počítá s tím, že zařízení sbírající data poskytuje tato data pro další zpracování ve vhodném formátu. V takovém případě je řetězec zpracování dat poměrně jednoduchý, protože Prometheus tak periodicky získává data prostřednictvím HTTP GET požadavků na koncový bod daného zařízení. Následně Grafana při dotazování Promethea, také pomocí HTTP GET požadavků, dostane data a ta se mohou vizualizovat a případně na základě uživatelem nastavených pravidel mohou spustit (díky webhooku z upozorňovacího systému Grafany) definovaný tok v Node-RED. Druhá varianta sběru dat není ideální, ale architektura je na ni připravená. Problém může totiž nastat v případě, že zařízení sbírající data nemůže data poskytovat ve vhodném formátu pro příjem dat Prometheem. V takovém případě je datový tok pozměněn v počáteční fázi, kdy data sbírá Node-RED a přeformátovává je do použitelného formátu. Zbytek datového toku je stejný jako v ideálním případě, což ukazuje na variabilitu architektury.



Obrázek 4.11: Finální podoba architektury

4.13 Závěr

V rámci své diplomové práce jsem vytvořil pokročilou a poměrně univerzální architekturu založenou na aplikacích Prometheus, Grafana a Node-RED, která je snadno použitelná pro velký počet různých scénářů, kdy je potřeba monitorovat větší počet zařízení. Po provedení analýzy otevřených monitorovacích systémů v kapitole 3 jsem tuto architekturu navrhl podle mnou definovaných konkrétních kritérií v podkapitole 3.9, které by takový systém měl podle mého úsudku splňovat. Kromě těchto kritérií jsem při návrhu kladl důraz na bezpečnost, distribuovatelnost, škálovatelnost a rozšiřitelnost celého řešení. Důvody pro zvolení mnou navržené architektury popisuji na začátku kapitoly 4. Výsledkem mé práce tedy není jedna konkrétní aplikace, ale předkládám komplexní pohled na složitou problematiku a na jeho základě navrhuji pokročilou architekturu, která má v dnešním rychle se rozvíjejícím světě internetu věcí velké potenciální využití. Kromě ukázkového simulovaného systému popisovaného v kapitole 4 jsem předložil argumenty pro zvolení mnou navrhované architektury pro použití v reálném systému.

Konkrétní výhody oproti jednodušším architekturám spočívají ve velmi snadné a efektivní škálovatelnosti a distribuovatelnosti. Architektura je velmi flexibilní ve sběru dat, protože je připravená na scénáře, kde zařízení neposkytují data v ideálním formátu. Tím je zajištěná robustnost a adaptabilita systému na různé typy datových zdrojů. Oproti jiným porovnávaným monitorovacím systémům disponuje tato architektura pokročilými možnostmi vizualizace, interakce s daty a jednoduché analýzy díky využití Grafany. Silnou stránkou je také možná implementace reakcí systému na potenciální problémy díky integraci Node-RED, a tudíž snadná automatizace procesů v systému. Tato architektura také umožňuje efektivně, díky využití aplikace Node-RED, snižovat dobu nefunkčnosti kritických komponent, a tudíž i snižovat náklady na provoz takového systému oproti systémům, které jsou zaměřeny pouze na monitoring a generování upozornění. Velká výhoda také spočívá v poměrně snadné a mnou podrobně popsané škálovatelnosti a přizpůsobitelnosti pro konkrétní použití v konkrétním systému. Dalším přínosem je i rozšíření mého vhledu do problematiky internetu věcí a posílení mých kompetencí v této oblasti, která je mi velmi blízká, protože v ní vidím velký potenciál pro budoucnost světa informačních technologií.

Architekturu, která byla navržena na základě případu užití monitoringu farmy 3D tiskáren, lze aplikovat na širokou škálu systémů, které vyžadují monitoring většího počtu zařízení. Obecně lze farmu tiskáren popsat jako soubor mnoha různých senzorů, jejichž data se v samotné 3D tiskárně spojují do určitých logických celků v krátkém časovém rámci. Takto lze popsat obecně velkou část IoT systémů, které na nejnižší úrovni sbírají data a předávají je do vyšších úrovní. Na konci tohoto řetězce je uživatel, který z daných dat vyvozuje závěry. Může nastat situace, která nebude z nějakých důvodů příznivá pro takový způsob sběru samotných dat, jaký je popsaný v mém případě užití, a tudíž nebude zahrnut v mnou navrhované architektuře. V takové situaci je velkou výhodou snadná rozšiřitelnost navrhované architektury o další komponenty, které mohou zajišťovat komunikaci s konkrétními zařízeními. Pokud by nevhodné podmínky spočívaly například v nestabilním spojení nebo nevhodnosti použití webového frameworku, tak lze problémy z toho vyplývající snadno eliminovat použitím například MQTT brokera mezi koncovými zařízeními a Prometheem. Pokud by nevhodné podmínky spočívaly v nepodporovaném nebo jinak nevhodném formátu, lze tento problém snadno řešit pomocí Node-RED tak, jak je to popsané v kapitole 4.

Výsledkem práce je tedy nejen ukázková aplikace mnou navržené architektury, ale i komplexní zpracování návrhu architektury, která je použitelná v mnoha různých scénářích monitoringu mnoha zařízení. Tato problematika je pro mě osobně velice zajímavá a předpokládám v budoucnosti pokračování rozmachu velkých sítí mnoha různých zařízení.

Použitá literatura

1. *IoT connected devices worldwide 2019-2030* [Statista] [online]. [cit. 2024-05-07]. Dostupné z: https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/.
2. *13 Best Open Source & Free Monitoring Tools* [DevopsCube] [online]. 2023-07-22. [cit. 2024-05-07]. Dostupné z: https://devopscube.com/best-opensource-monitoring-tools/.
3. METRICFIRE. *Top 5 Open Source Server Monitoring Tools* [online]. [cit. 2024-05-07]. Dostupné z: https://www.metricfire.com/blog/top-5-open-source-server-monitoring-tools/.
4. BERMAN, Daniel. *A Guide to Open Source Monitoring Tools* [Logz.io] [online]. 2019-07-11. [cit. 2024-05-07]. Dostupné z: https://logz.io/blog/open-source-monitoring-tools/.
5. HERNANTES, Josune; GALLARDO, Gorka; SERRANO, Nicolás. IT Infrastructure-Monitoring Tools. *IEEE Software* [online]. 2015, roč. 32, č. 4, s. 88–93 [cit. 2024-05-07]. ISSN 1937-4194. Dostupné z DOI: 10.1109/MS.2015.96. Conference Name: IEEE Software.
6. HERNANTES, Josune; GALLARDO, Gorka; SERRANO, Nicolás. IT Infrastructure-Monitoring Tools. *IEEE Software* [online]. 2015, roč. 32, č. 4, s. 88–93 [cit. 2024-05-08]. ISSN 0740-7459, ISSN 1937-4194. Dostupné z DOI: 10.1109/MS.2015.96.
7. *About : Node-RED* [online]. [cit. 2024-05-08]. Dostupné z: https://nodered.org/about/.
8. PROMETHEUS. *Overview | Prometheus* [online]. [cit. 2024-05-08]. Dostupné z: https://prometheus.io/docs/introduction/overview/.

9. *Official Elasticsearch Pricing: Elastic Cloud, Managed Elasticsearch* [online]. [cit. 2024-05-08]. Dostupné z: https://www.elastic.co/pricing.

10. *What is ELK stack? - Elasticsearch, Logstash, Kibana Stack Explained - AWS* [Amazon Web Services, Inc.] [online]. [cit. 2024-05-08]. Dostupné z: https://aws.amazon.com/what-is/elk-stack/.

11. *What is Elasticsearch? - Elasticsearch Explained - AWS* [Amazon Web Services, Inc.] [online]. [cit. 2024-05-08]. Dostupné z: https://aws.amazon.com/what-is/elasticsearch/.

12. *History of Nagios | Nagios Open Source* [online]. [cit. 2024-05-08]. Dostupné z: https://www.nagios.org/about/history/.

13. *Overview | Nagios Open Source* [online]. [cit. 2024-05-08]. Dostupné z: https://www.nagios.org/about/overview/.

14. PROMETHEUS. *Writing exporters | Prometheus* [online]. [cit. 2024-05-08]. Dostupné z: https://prometheus.io/docs/instrumenting/writing_exporters/.

15. *A Complete Overview of the Best Data Visualization Tools | Toptal®* [Toptal Design Blog] [online]. [cit. 2024-05-08]. Dostupné z: https://www.toptal.com/designers/data-visualization/data-visualization-tools.

16. *Top Data Visualisation Tools For 2024* [Logit.io] [online]. [cit. 2024-05-08]. Dostupné z: https://logit.io.

17. JAYAWARDHANA, Shamal. *Best Data Analytics Tools: Comparing Grafana vs. Yellowfin* [Medium] [online]. 2023-10-09. [cit. 2024-05-08]. Dostupné z: https://medium.com/@m.shamal.j/best-data-analytics-tools-comparing-grafana-vs-yellowfin-cd9ef11ac7fa.

18. *Roles and permissions | Grafana documentation* [Grafana Labs] [online]. [cit. 2024-05-08]. Dostupné z: https://grafana.com/docs/grafana/latest/administration/roles-and-permissions/.

19. *Alerting | Grafana documentation* [Grafana Labs] [online]. [cit. 2024-05-08]. Dostupné z: https://grafana.com/docs/grafana/latest/alerting/.

20. SHAN, Yang Guo et al. Research on Monitoring of Information Equipment Based on Zabbix for Power Supply Company. In: *2021 3rd International Conference on Applied Machine Learning (ICAML)* [online]. Changsha, China: IEEE, 2021, s. 487–491 [cit. 2024-05-08]. ISBN 978-1-66542-125-6. Dostupné z DOI: 10.1109/ICAML54311.2021.00108.

21. SQUADCAST. *Zabbix vs Prometheus: Choosing the Right Monitoring Tool for Your Needs* [Medium] [online]. 2024-04-18. [cit. 2024-05-08]. Dostupné z: https://medium.com/@squadcast/zabbix-vs-prometheus-choosing-the-right-monitoring-tool-for-your-needs-9c3177daf21f.

22. INOUYE, Jenna. *Prometheus vs Zabbix | Network Monitoring Tools Comparison* [TechRepublic] [online]. 2022-04-15. [cit. 2024-05-08]. Dostupné z: https://www.techrepublic.com/article/prometheus-vs-zabix/.

23. *Prometheus vs. Elasticsearch* [Atatus Blog - For DevOps Engineers, Web App Developers and Server Admins.] [online]. 2024-03-08. [cit. 2024-05-08]. Dostupné z: https://www.atatus.com/blog/prometheus-vs-elasticsearch/.

24. METRICFIRE. *Prometheus vs. ELK* [online]. [cit. 2024-05-08]. Dostupné z: https://www.metricfire.com/blog/prometheus-vs-elk/.

25. *What is a webhook?* [online]. [cit. 2024-05-08]. Dostupné z: https://www.redhat.com/en/topics/automation/what-is-a-webhook.

26. PROMETHEUS. *Querying basics | Prometheus* [online]. [cit. 2024-05-08]. Dostupné z: https://prometheus.io/docs/prometheus/latest/querying/basics/.

27. *Docker monitoring* [Grafana Labs] [online]. [cit. 2024-05-08]. Dostupné z: https://grafana.com/grafana/dashboards/15798-docker-monitoring/.

28. BHAT, Tanmay. *tanmay-bhat/grafana-dashbaords* [online]. 2023. [cit. 2024-05-11]. Dostupné z: https://github.com/tanmay-bhat/grafana-dashbaords. original-date: 2022-03-14T05:31:40Z.

29. MOTA, Levi et al. A Comparative Analysis of Network Management Protocols in IoT Applications. *Journal of Computer Science*. 2018, roč. 14, s. 1238–1246. Dostupné z DOI: 10.3844/jcssp.2018.1238.1246.

30. MOTA, Levi Costa; MORENO, Edward David; RIBEIRO, Admilson Lima. A comparative analysis of protocols for IoT network management. In: *Proceedings of the Euro American Conference on Telematics and Information Systems* [online]. New York, NY, USA: Association for Computing Machinery, 2018, s. 1–5 [cit. 2024-05-08]. EATIS '18. ISBN 978-1-4503-6572-7. Dostupné z DOI: 10.1145/3293614.3293646.

31. *Loki overview | Grafana Loki documentation* [Grafana Labs] [online]. [cit. 2024-05-08]. Dostupné z: https://grafana.com/docs/loki/latest/get-started/overview/.