Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Systém pro sběr a analýzu crash reportů pro Unreal Engine

2024

Jakub Štarman

Vedoucí práce:
Mgr. Radek Jánoštík Ph.D.

Studijní program: Informační technologie,
kombinovaná forma

Bibliografické údaje

Autor:	Jakub Štarman
Název práce:	Systém pro sběr a analýzu crash reportů pro Unreal Engine
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2024
Studijní program:	Informační technologie, kombinovaná forma
Vedoucí práce:	Mgr. Radek Jánoštík Ph.D.
Počet stran:	49
Přílohy:	elektronická data v úložišti katedry informatiky
Jazyk práce:	český

Bibliographic info

Author:	Jakub Štarman
Title:	System for Collecting and Analyzing Crash Reports for Unreal Engine
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2024
Study program:	Information Technologies, combined form
Supervisor:	Mgr. Radek Jánoštík Ph.D.
Page count:	49
Supplements:	electronic data in the storage of department of computer science
Thesis language:	Czech

Anotace

Systém, který byl navržen a vyvinut pro sběr, analýzu a vizualizaci chybových hlášení (crash reportů) pro projekty v Unreal Engine na platformě Windows. Poskytuje vývojářům nástroj pro efektivní identifikaci, analýzu a řešení chyb, čímž zvyšuje stabilitu a kvalitu projektů. Systém zahrnuje moduly pro sběr dat, jejich zpracování, analýzu a webové rozhraní pro vizualizaci. Během testování byl systém ověřen na několika tisících událostech z reálného herního vývoje, což potvrdilo jeho schopnost správně identifikovat a kategorizovat problémy. Navrhovaná vylepšení cílí na jeho plnohodnotné nasazení do praxe.

Synopsis

A system designed and developed for the collection, analysis, and visualization of crash reports for projects in Unreal Engine on the Windows platform. It provides developers with tools for efficient identification, analysis, and resolution of errors, thereby enhancing the stability and quality of projects. The system includes modules for data collection, processing, analysis, and a web interface for visualization. During testing, the system was verified with several thousand events from real game development, confirming its ability to accurately identify and categorize issues. Proposed enhancements aim for its full deployment in practice.

Klíčová slova: unreal engine; crash; crash report; crash reporter; vývoj her

Keywords: unreal engine; crash; crash report; crash reporter; game development

Na tomto místě bych rád vyjádřil poděkování Mgr. Radku Janoštíkovi, Ph.D., vedoucímu mé bakalářské práce, za příkladné vedení, doporučení a věcné připomínky. Dále děkuji panu Rákosníčkovi za průběžnou zpětnou vazbu. Poděkování patří také mým kolegům za odborné rady a připomínky a v neposlední řadě své přítelkyni za podporu.

Odevzdáním tohoto textu jeho autor/ka místopřísežně prohlašuje, že celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

Obsah

Seznam tabulek

1 Úvod

Tvorba videoher je komplexní proces, který zahrnuje mnoho odvětví informatiky. Jedná se o proces, v rámci kterého se vývojáři setkávají s mnoha výzvami, od grafického designu a programování až po testování a uživatelskou podporu. Herní průmysl již dávno překročil hranice zábavy a stal se vážným oborem, v němž inovace a technické dovednosti hrají klíčovou roli. Vývoj her je dynamický a neustále se rozvíjející sektor, kde každý nový projekt přináší řadu technických i kreativních problémů k řešení.

Chyby a problémy jsou v tomto rychle se vyvíjejícím odvětví nevyhnutelné. Ty se liší od drobných kosmetických chyb, které neovlivní hráčský zážitek, po technické problémy, které mohou způsobit pády her (*crash*) a tím narušit uživatelskou zkušenost. Práce se zaměřuje na druhý zmíněný typ chyb – tedy ty, které mohou přerušit hru a představují kritické situace vyžadující rychlou a efektivní odezvu od vývojářů. Vzniká tak potřeba tyto chyby sbírat a analyzovat.

V této bakalářské práci se věnuji technickým aspektům sběru a analýzy *crash reportů*, konkrétně v kontextu herního enginu Unreal Engine na platformě Windows, který je široce uznáván a využíván ve světě primárně k vývoji her.

Crash report je záznam chybné události. Obsahuje klíčové informace pro identifikaci a řešení závažných problémů, a proto je důležité, aby vývojáři disponovali efektivními nástroji pro jejich sběr a analýzu. Tato potřeba vyplynula z mé praxe vývojáře v herním průmyslu.

Cílem této práce je navrhnout a implementovat systém, který umožní vývojářům rychle nalézat, analyzovat a řešit příčiny chyb, což by mělo vést k vyšší stabilitě a lepší kvalitě herních projektů.

2 Motivace

Motivací pro vytvoření této bakalářské práce byl text v dokumentaci „Crash Reporting Unreal Enginu“ [1], který upozorňuje na nutnost přípravy serveru schopného přijímat a analyzovat chybové zprávy od vzdálených klientů pro plné využití komponenty CrashReporting. Unreal Engine ve svých zdrojových kódech tento server nenabízí a odkazuje na existující řešení od třetích stran.

Existující systémy pro sběr a analýzu událostí obvykle podporují sběr dat z různých aplikací a enginů. Tyto systémy však často zaznamenávají pouze fakt, že událost nastala, a seskupují události pomocí zásobníku volání.

Žádný ze systémů doporučených v Unreal Engine Dokumentaci [2] však nenabízí specifické rozdělení událostí, které by bylo výhodné pro sběr chyb během vývoje v Unreal Enginu (kapitola 3.2), kde je zvláště užitečné rozlišovat mezi událostmi typu Crash, Assert a Ensure popsanými v kapitole 3.2.9. Navíc je zde velká výhoda mít rychlý přístup k chybové zprávě generované kontrolním makrem (kapitola 3.2.8) nebo zprávě od uživatele, u kterého došlo k události nebo havárii aplikace. U některých systémů je sice možné tyto informace dohledat, ale často to vyžaduje použití vlastních dotazů[1] nebo je to obtížně vyhledatelné.

Tato řešení vyhovují potřebám pro sběr a analýzu uživatelských (hráčských) crash reportů, nikoliv však pro potřeby vývoje. Z těchto důvodů v mé profesní praxi vyvstala potřeba vytvořit specializovaný systém.

[1] Dotazy podobné dotazům v jazyce SQL

3 Vysvětlení problematiky

V této sekci budou definovány pojmy a technologie, které se týkají této bakalářské práce. Nejdříve popíšeme pojem herního enginu jako takového, poté se podrobněji zaměříme na Unreal Engine a vývoj v něm, a nakonec na funkcionalitu jeho systému CrashReporting. Na závěr představíme použité technologie a nástroje.

3.1 Herní Engine

Jedná se o ekosystém nástrojů a technologií určený pro vývoj videoher. Jeho hlavním účelem je ulehčit práci vývojářům tím, že jim poskytuje jakési jádro hry, které řeší základní funkcionalitu jako je třeba zpracování uživatelského vstupu, vykreslování grafiky a fyzikální engine. Vývojáři tedy nemusí tyto komponenty znovu vyvíjet. První enginy vznikaly oddělením technického jádra hry od konkrétního herního obsahu. Tato jádra bylo možné dále znovu použít pro tvorbu jiných her většinou formou licencování.

3.2 Unreal Engine

Unreal Engine, vyvinutý společností Epic Games, byl poprvé vydán v roce 1998 s primárním cílem podpořit vývoj první hry ze série Unreal. Během vývoje hry v roce 1996 bylo však již zahájeno licencování třetím stranám, což bylo klíčové pro ekonomické přežití Epic Games [3].

V době psaní práce je Unreal Engine jedním z nejrozšířenějších a technologicky pokročilých herních enginů, který nachází uplatnění nejen ve videohrách, ale i v filmové produkci, televizním vysílání, architektuře, automobilovém průmyslu a ve vizualizacích pro virtuální realitu.

Unreal Engine je dostupný pod vlastní licencí [4], která umožňuje přístup k jeho zdrojovým kódům, což vývojářům a výzkumníkům poskytuje mimořádnou flexibilitu ve vývoji a přizpůsobení enginu jejich projektům. Společnost Epic Games podporuje otevřenou komunitu, kde mohou uživatelé přispívat k vylepšení enginu prostřednictvím pull requestů[5] na GitHub[6] repozitáři.

3.2.1 Unreal Editor a struktura projektu

Unreal Editor je klíčovou komponentou Unreal Engine. Jedná se o integrované vývojářské prostředí umožňující vývojářům tvořit a upravovat obsah v reálném čase. Nabízí rozmanité nástroje pro animaci, skládání světů, skriptování a další nástroje pro ulehčení vývoje.

Každý projekt v Unreal Engine se skládá z engine a herní části, což umožňuje efektivní oddělení vývoje hry od komponent enginu. Tato struktura je zásadní pro pochopení různých konfigurací a režimu spuštění.

3.2.2 Režimy spuštění: Editor nebo Game

Unreal Engine nabízí dva hlavní režimy spuštění: Editor a Game. V režimu Editor je aktivován celý Unreal Editor, což vývojářům umožňuje využít veškeré nástroje a funkce enginu pro tvorbu a testování herního obsahu. Tento režim je využíván vývojáři pro interaktivní tvorbu a modifikace herních světů a prvků.

V režimu Game se naopak spouští pouze herní kód spolu s nezbytnými funkcemi jádra Unreal Engine, čímž se simulují podmínky, ve kterých budou hráči hru skutečně spouštět. Tento režim je zásadní pro finální testování a optimalizaci, neboť poskytuje autentický zážitek z konečného herního prostředí na cílových platformách.

Tato struktura v režimech spuštění spojená s využitím jednotlivých konfigurací umožňuje vývojářům efektivně ladit projekt během vývoje, zatímco zajišťuje, že konečný produkt nebude jakkoliv zatížen pozůstatky vývojářských nástrojů.

3.2.3 Konfigurace sestavení projektů Unreal Engine

Konfigurace sestavení v Unreal Engine jsou optimalizovány pro různé fáze vývoje a distribuce projektů. Existují vývojové a distribuční konfigurace, každá s určeným účelem a nastavením. Unreal Engine rozlišuje mezi konfiguracemi pro engine a samotný projekt hry, což umožňuje různé úrovně ladění a optimalizace.

3.2.3.1 Vývojové konfigurace Vývojové konfigurace jsou navrženy pro různé potřeby během vývoje a testování:

- **Debug** - Tato konfigurace je bez optimalizací jak herního tak i engine kódu, obsahuje nejvíce ladících informací a je ideální pro ladění složitých chyb.
- **Development** - Běžně používaná pro vývoj a testování, nabízí vyvážený kompromis mezi optimalizací a možnostmi ladění.
- **DebugGame** - Hybridní konfigurace, kde je Engine kompilován v konfiguraci Development a herní kód v konfiguraci Debug. Toto umožňuje podrobné ladění herního kódu bez výrazného snížení výkonu způsobeného neoptimalizovaným kódem enginu.
- **Test** - Podobná distribuční konfiguraci *Shipping*, ale zachovává některé testovací a ladící schopnosti, například logování. Tato konfigurace je určena pro vnitřní testování před finálním vydáním.

3.2.3.2 Distribuční konfigurace Pro distribuci a finální vydání hry se používá následující konfigurace:

- **Shipping** - Nejvíce optimalizovaná konfigurace určená pro koncového zákazníka. Maximálně zvyšuje výkon a odstraňuje veškeré ladící informace.

3.2.4 Vývoj kódu v Unreal Engine

Vývoj kódu v Unreal Engine se liší oproti klasickému vývoji C++. Unreal Engine používá modifikovanou verzi C++, která je označována jako „Unreal C++“. Tento jazyk zahrnuje několik vlastních rozšíření a zjednodušení, která jsou určena k podpoře efektivnějšího vývoje her. Na rozdíl od standardního C++ nabízí Unreal C++ integrovanou podporu pro správu paměti a reflexi, což jsou prvky, které nejsou typické pro standardní C++. Tyto funkce usnadňují správu herních objektů a tříd v rámci Unreal Engine, což zmenšuje zátěž vývojářů spojenou se správou paměti a dynamickou informací o typech.

Unreal C++ rozšiřuje standardní C++ o řadu maker, specifikátorů vlastností a vestavěných tříd, které jsou speciálně navrženy pro zjednodušení běžných úloh ve vývoji her. Těmi může být správa vlastnictví objektů v paměti, multiplayerové funkce a tvorba uživatelského rozhraní. Tento přístup činí programování v Unreal C++ odlišným od typického programování v C++, kde vývojáři musí manuálně zvládat mnoho úloh na nízké úrovni, které Unreal C++ automatizuje nebo zjednodušuje. Unreal Engine poskytuje rozsáhlou dokumentaci o tom jak vyvíjet kód [7].

3.2.5 Kompilace

Proces kompilace kódu v Unreal C++ se odlišuje od tradičního kompilování tím, že se během procesu kompilace spustitelných souboru provádí kroky specifické k Unreal C++, kterými se nastaví prostředí pro jejich kompilace podle platformy a nastavené konfigurace. Toto například ovlivňuje předdefinování některých maker[2], přítomnost kontrolních maker a optimalizační úroveň jednotlivých sestavení.

Během kompilace vznikají symbolové soubory. Tyto soubory obsahují přídavné informace o spustitelném souboru, potřebném k ladění. Tyto symbolové soubory se při finální distribuci nedodávají koncovému hráči. Je tedy důležité pro vývojáře tyto soubory ukládat, aby v případě chybové události byli schopni odladit příčinu.

3.2.6 Ladění

Ladění projektů v Unreal Enginu je podobné ladění jakékoliv jiné C++ aplikace. Použití vývojových nástrojů, které podporují Unreal Engine, významně ulehčuje ladění, neboť tyto nástroje umožňují zobrazovat struktury specifické pro Unreal C++. Ladit je možné jak aplikaci běžící v reálném čase, tak její *minidump*. Pro ladění jsou nezbytné symboly, které mohou být uložené lokálně na stroji vývojáře, nebo získané vzdáleně přes symbol server.

[2]Makra, pro možnost vyloučení kódu pro editor, specifické platform a další

3.2.7 Minidump

Minidump je zmenšený výpis paměti aplikace, který se vytváří v momentě jejího neočekávaného ukončení nebo pádu. Obsahuje klíčové informace nutné pro analýzu problému, jako jsou registry procesoru, zásobník volání funkcí a informace o běžících vláknech. Díky své malé velikosti je snadno odesílatelný od uživatele k vývojáři, který jej může využít pro identifikaci a odstranění chyby v aplikaci. Minidump umožňuje efektivní diagnostiku problémů bez nutnosti přístupu k plnému výpisu paměti aplikace.

3.2.8 Kontrolní makra v Unreal Engine

Unreal C++ nabízí několik možností, jak detekovat a diagnostikovat nečekané nebo nevalidní situace během provádění programu. Jedním z těchto mechanizmů jsou kontrolní makra [8]. Ta jsou často používaná v situacích, jako je třeba ochrana přístupu mimo rozsah alokovaného pole, kontrola správnosti rekurzivní funkce nebo ověření, že se neprovádí žádná neimplementovaná část kódu. Klíčovou vlastností kontrolních maker je to, že v kódu určeném pro distribuci neexistují. Nemají tím pádem žádný vliv na výkon finálního produktu.

Unreal Engine nabízí 3 typy kontrolních maker. Těmi jsou:

3.2.8.1 Check (Zkontroluj)

Makro, které se nejvíce podobá C++ assertu. Zastaví průběh programu pokud se výraz v prvním parametru vyhodnotí na nepravdu.
Následující verze makra jsou dostupná pro použití (tabulka 1).

Název makra	Parametry	Užití
check/checkSlow	Testovaný výraz	Kontrola pravdivosti výrazu
checkf/checkfSlow	Testovaný výraz Formátovaná zpráva	Kontrola pravdivosti výrazu Zobrazení zprávy
checkCode	Blok kódu	Provede blok kódu
checkNoEntry	(Bez parametrů)	Kontrola zakázaného vstupu
checkNoReentry	(Bez parametrů)	Kontrola opakovaného vstupu
checkNoRecursion	(Bez parametrů)	Kontrola rekurze
unimplemented	(Bez parametrů)	Pro neimplementované funkce

Tabulka 1: Popis typů Check maker

Příklad použití těchto maker jsou ve zdrojových kódech 1, 2, 3, 4.
Check makra se provádí ve všech konfiguracích editoru, ale pouze ve vývojových konfigurací hry (Debug, Development). Makra označena jako „Slow“ fungují jen v konfiguracích Debug.

```
1  // Tato funkce by nikdy neměla být volána s nulovým parametrem
       JumpTarget. Je potřeba zastavit program pokud se to stane
2  void AMyActor::CalculateJumpVelocity(AActor* JumpTarget, FVector&
       JumpVelocity)
3  {
4     check(JumpTarget != nullptr);
5     // Na výpočet potřebujeme znát cíl. Pomocí toho makra si můžeme b
          ýt jistí, že parametr JumpTarget nebude nulový
6  }
```

Zdrojový kód 1: Ukázka použití makra Check v Unreal C++

```
1  // Tento kód zastaví prováděni programu, pokud přidáme novou hodnotu
       do výčtového typu, ale zapomeneme ji přidat do konstrukce
       switch
2  switch (MyShape)
3  {
4     case EShapes::S_Circle:
5        // (Handle circles.)
6        break;
7     case EShapes::S_Square:
8        // (Handle squares.)
9        break;
10    default:
11       // Pro každou hodnotu výčtového typu by měl existovat case,
             tato situace by nikdy neměla nastat.
12       checkNoEntry();
13       break;
14 }
```

Zdrojový kód 2: Ukázka použití makra checkNoEntry v Unreal C++

Tato makra jde zapnout i v ostatních konfiguracích, pokud existuje podezření, že kód uvnitř makra mění hodnotu, nebo existují chyby u kterých máme podezření, že jsou odhalitelné těmito makry.
Upravuje se úpravou definice makra `USE_CHECKS_IN_SHIPPING` na `1`.
Projekty by měli být distribuovány s výchozí hodnotou toho makra na `0`.

```
// Tento objekt má testovací funkci, IsEverythingOK, která nemá
// žádné vedlejší účinky, ale v případě problému vrací nepravdu.
// Pokud tato situace nastane, provádění programu končí fatální
    chybou.
// Jelikož tento kód nemá žádný vedlejší účinky a pouze slouží k
    diagnostickým účelům, není potřeba jej provádět v distribučních
    konfiguracích.
checkCode(
if (!IsEverythingOK())
{
   UE_LOG(LogCategory, Fatal, TEXT("Something is wrong with %s!
       Terminating."), *GetFullName());
}
);
```

Zdrojový kód 3: Ukázka použití makra checkCode v Unreal C++

```
// Ve spojovém seznamu by nikdy neměla být smyčka.
// Program by se v takovém případě zacyklil.
// Kontrola toho, že v seznamu není cyklus je ovšem pomalá operace.
// Chceme tedy tuto kontrolu provádět pouze v Debug konfiguracích.
checkfSlow(!MyLinkedList.HasCycle(), TEXT("Found a cycle in the list
    !"));
// (Normální zbytek kódu průchodu spojového seznamu)
```

Zdrojový kód 4: Ukázka použití makra checkfSlow v Unreal C++

3.2.8.2 Verify (Ověř)

Verify makro je podobné s makrem Check. Rozdíl je v tom, že výraz se vyhodnocuje v každé konfiguraci. Má tedy vedlejší účinky a mělo by být použito pouze v případě, kdy se testovaný výraz musí provést nezávisle od diagnostických kontrol (zdrojový kód 5).
Dostupná Verify makra pro použití:

Název makra	Parametry	Užití
verify/verifySlow	Testovaný výraz	Kontrola pravdivosti výrazu
verifyf/verifyfSlow	Testovaný výraz Formátovaná zpráva	Kontrola pravdivosti výrazu Zobrazení zprávy

Tabulka 2: Popis typů Verify maker

Stejně jako makro Check, makro Verify zastaví průběh programu ve všech konfiguracích editoru, ale pouze ve vývojových konfiguracích hry (Debug, Development). Makra označená jako „Slow“ fungují jen v konfiguracích Debug.

Definice `USE_CHECKS_IN_SHIPPING` na `1` mění chování makra Verify na makro Check. V ostatních případech se makro Verify chová tak, že provádí vždy

```
1 // Tento kód načítá knihovnu. Pokud se knihovna nenačte, průběh
    programu se zastaví
2 // Využíváme zde toho, že makro má vedlejší účinky tím, že
    vyhodnocuje výraz pokaždé
3 verify(Plugin->LoadLibrary(LibraryName) == false);
```

Zdrojový kód 5: Ukázka použití makra Verify v Unreal C++

vyhodnocení výrazu, ale nezastavuje průběh.

3.2.8.3 Ensure (Zajisti)

Ensure makro je poslední z kontrolních maker, které Unreal C++ nabízí. Narozdíl od Check a Verify maker, makro Ensure nezastavuje průběh programu v případě, že jeho vstupní výraz se vyhodnotí na nepravdu. Místo toho se informuje CrashReporting systém o události a poté se umožní programu pokračovat.

V případě, že je v projektu zapnutý CrashReporter, nahlásí tuto událost v *Unattended* režimu. Nahlášení může způsobit „zamrznutí“ aplikace, jelikož CrashReporter čeká, až se záznam o události odešle na server.Toto neplatí během ladění, kdy ensure, zastaví průběh programu, aby vývojář mohl zanalyzovat jeho příčinu.

Ensure provádí nahlášení pouze jednou během života aplikace, aby se omezilo množství událostí, které mohou vzniknout použitím tohoto makra a které by mohly zahltit server. Pokud je i přesto požadavek aby se nahlášení provedlo pokaždé, je potřeba použít varianty makra **ensureAlways** a **ensureAlwaysMsgf**.

Ensure makro je užitečné pro zachycení neočekávaných situací, které sice nejsou fatální, ale mohou způsobovat potenciální problém. Příkladem může být třeba zachycení situace, kdy funkce navrátí neočekávanou hodnotu, která by za normálních okolností neměla nastat.

Zejména využití makra ensure se zprávou (ensureMsgf) může být vhodné pro předání informací přímo přes text výjimky (ukázka použití v kódu 6), která nastala, jako třeba stav testovaného systému, aby se tento stav nemusel zapisovat do logu a pak v případě události dohledávat. Dostupná Ensure makra pro použití:

Název makra	Parametry	Užití
ensure	Testovaný výraz	Ověří předpoklad
ensureMsgf	Testovaný výraz Formátovaná zpráva	Kontrola pravdivosti výrazu Zobrazení zprávy

Tabulka 3: Popis typů ensure maker

Testovaný výraz se vždy vyhodnocuje, ale informuje CrashReporting systém

```
1  // Tato ukázka použití makra odhaluje pouze malou chybu, která může
       nastat v distribuční konfiguraci.
2  // Malá velikost chyby nám umožňuje předejít nevalidní situaci a
       pokračovat v provádění programu.
3  // Můžeme si myslet, že jsme opravili příčinu chybné situace, ale
       pro budoucí případ chceme být informování, že chyba nastala.
4  void AMyActor::Tick(float DeltaSeconds)
5  {
6     Super::Tick(DeltaSeconds);
7     // Zajisti, že hodnota bWasInitialized je pravdivá. Pokud není,
          zaloguje že chyba stále nebyla opravena
8     if (ensureMsgf(bWasInitialized, TEXT("%s ran Tick() with
          bWasInitialized == false"), *GetActorLabel()))
9     {
10       // (Kód který potřebuje pravdivou testovanou hodnotu)
11
12    }
13 }
```

Zdrojový kód 6: Ukázka použití makra ensureMsgf v Unreal C++

jen ve vývojových konfiguracích hry (Debug, Development, Test) a ve všech konfiguracích editoru.

3.2.9 CrashReporting systému v Unreal Enginu

CrashReporting systém v Unreal Enginu představuje klíčový nástroj pro zachycování a zpracování informací o chybách a haváriích aplikací. Tento systém se skládá ze dvou hlavních komponent: integrované funkcionality v Unreal Enginu a samostatné aplikace zvané CrashReporter.

První část, integrovaná přímo do Unreal Engine, poskytuje hře možnost zapisovat dodatečný kontext o dění v průběhu provádění herního kódu (zdrojový kód 7), což pomáhá vývojářům lépe porozumět okolnostem pádu.

```
1 void OnEnterMyGameMode()
2 {
3     FPlatformCrashContext::SetGameData(TEXT("GameMode"), TEXT("
          MyGameModeName"));
4 }
5 void OnExitMyGameMode()
6 {
7     FPlatformCrashContext::SetGameData(TEXT("GameMode"), FString());
8 }
```

Zdrojový kód 7: Ukázka zapsání dodatečného herního kontextu

Nejčastější události zpracovávané tímto systémem zahrnují *Crash*, *Assert* a *Ensure*, přičemž každá má specifický dopad na chování aplikace (tabulka 4):

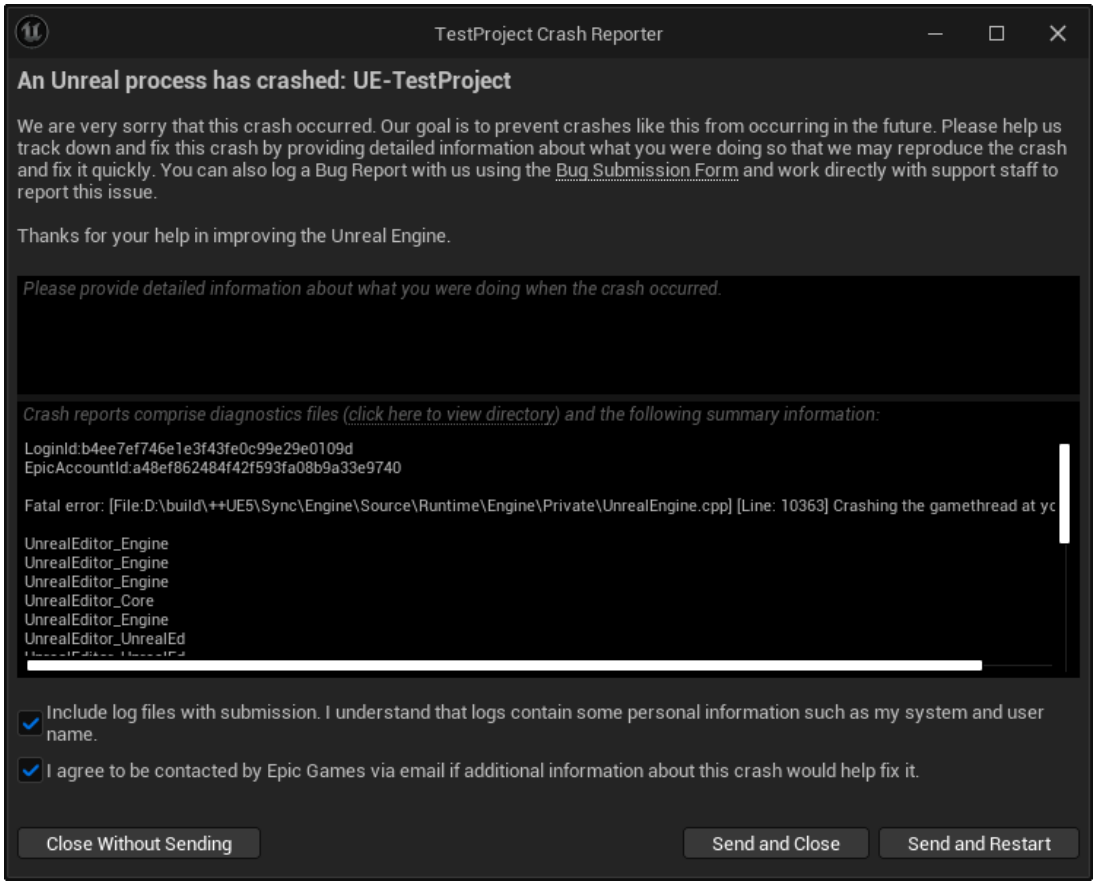
Typ události	Popis
Crash	Hávarie aplikace způsobená kritickou chybou
Assert	Hávarie vyvolaná kontrolními mechanizmy
Ensure	Kontrolovaný výraz nebyl splněn, provádění pokračuje

Tabulka 4: Nejčastější typy událostí zpracovávaných CrashReporting systémem

Druhá klíčová komponenta, aplikace CrashReporter, je distribuována s verzemi editoru nebo hry. Tato aplikace umožňuje zaslat informace o pádu na server, kde jsou data dále ukládána pro analýzu.
CrashReporter může pracovat ve dvou režimech:

1. **Attended**: Když aplikace havaruje, CrashReporter zobrazí dialogové okno (obrázek 1), které uživatele informuje o chybě a nabízí mu možnost poslat záznam o události nebo ho ignorovat.

2. **Unattended**: V tomto režimu CrashReporter automaticky odesílá záznamy o události na server bez uživatelské interakce, což je ideální pro serverové aplikace nebo když uživatelské rozhraní není dostupné.



Obrázek 1: Příklad vzhledu okna CrashReportéru

Infrastruktura CrashReporting systému se tedy skládá nejen z nástroje na zachytávání a odesílání informací o pádech, ale také serveru pro zpracování těchto dat, umožňující vývojářům rychle reagovat na problémy a zlepšovat stabilitu aplikace.

Adresa serveru, kam se události posílají, je nastavitelná pomocí změny konfiguračního souboru, a konfigurace samotné aplikace CrashReporter může být upravena v nastavení projektu, aby odpovídala specifickým potřebám projektu.

3.2.10 Chování CrashReporting systému při události

Při události systém CrashReporting vytvoří záznam, který je identifikován pomocí dvou identifikátorů (tabulka 5). Prvním identifikátorem je *GUID samotné události* (dále v textu označováno jako „GUID“), který se skládá z prefixu závislého na verzi Unreal Engine, platformě a GUID čísla. Druhým identifikátorem je *Execution GUID*, což je jednoznačný identifikátor instance běhu aplikace. Jelikož může více událostí sdílet jeden Execution GUID, umožňuje to sledovat všechny události v rámci jednoho spuštění aplikace.

Typ GUID	Hodnota
GUID události	UE4CC-Windows-21A429D3780...2BE61D95551_0000
Execution GUID	21E978094BE672EE9D3354A1B2D55199

Tabulka 5: Příklad hodnot jednotlivých GUID

Dále se provedou následující kroky:

1. Sesbírání důležitých informací o systému, aplikaci a platformě.
2. Vytvoření .dmp souboru pomocí Windows API.

GUID slouží k vytvoření složky, do které CrashReporter uloží soubory:

- CrashContext.runtime-xml - Kontextový soubor s sesbíranými informacemi.
- CrashReportClient.ini - Konfigurační soubor klienta.
- *.log - Aplikační log, pokud aplikace zaznamenává logy.
- *.dmp - Minidump záznam aplikace v době události.

Tato složka se ukládá do umístění Saved/Crashes/ ve složce aplikace.

V případě, že CrashReporter odesílá záznam na server, používá k tomu HTTP protokol s metodou POST. CrashReporter v URL pošle pomocí parametrů krátký souhrn o aplikaci CrashReporter a o uživateli, který požadavek odeslal. Do těla požadavku vkládá obsah složky, který je zaserializován do bajtů a zkomprimován pomocí metody Zlib.

3.2.11 Informace obsažené v záznamu CrashReporting systému při události

Každý záznam obsahuje XML soubor s informacemi o události ve strojově čitelném formátu, rozdělený do sekcí.
Ze sekce „RuntimeProperties“ nás zajímají hlavně (tabulka 6):

Jméno v XML	**Meaning**
CrashGUID	GUID
ExecutionGuid	Execution GUID události
IsEnsure	Událost byla vyvolána makrem Ensure
IsAssert	Událost byla vyvolána Assertem
CrashType	Typ události
CommandLine	Příkaz kterým byla aplikace spuštěna
ErrorMessage	Chybová zpráva události
SecondsSinceStart	Počet sekund od zapnutí aplikace
GameName	Jméno spuštěné aplikace
ExecutableName	Jméno spustitelného souboru aplikace
BuildConfiguration	Konfigurace ve které je aplikace sestavena
PlatformName	Jméno platformy na které aplikace běží
EngineMode	Mód enginu ve kterém běžel v době události
LanguageLCID	Jazyk aplikace ve které běží
BaseDir	Cesta ke složce ze které aplikace byla spuštěna
UserDescription	Komentář od uživatele
MachineId	ID stroje, na které aplikace byla spuštěna
EpicAccountId	ID Epic účtu, uživatele, u kterého událost nastala
DeploymentName	ID aplikace na obchodní platformě Epic Store
Misc.*	Informace o stroji (CPU, OS)
MemoryStats.*	Informace o paměti a jejím využití aplikací
CallStack	Informace k rychlému ladění
NumMinidumpFramesToIgnore	Rámce pro přeskočení v minidumpu
TimeOfCrash	Čas na systému v době události
bAllowToBeContacted	Informace, uživatele o souhlasu s kontaktováním

Tabulka 6: Zajímavé položky ze sekce Runtime Properties

Kromě toho zde najdeme sekce věnované platformě, které poskytují informace o jejím stavu, jako je například stav baterie.

Další sekce zahrnují informace o enginu, například o použitém vykreslovacím rozhraní, a sekce, které vyplňuje přímo hra. Tyto dvě poslední zmíněné sekce jsou aktivní pouze v režimu Game.

3.3 Použité technologie a nástroje

V této části se zaměříme na klíčové technologie a nástroje, které byly použity během vývoje projektu. Mezi tyto nástroje patří programovací jazyk Python [**empty citation**][9], webový framework FastAPI, frontend framework Bootstrap, šablonovací engine Jinja2, webový server NGINX, symbol server, nástroj minidump-stackwalk a platforma Docker.

3.3.1 FastAPI

FastAPI [10] je mladý, moderní a stabilní webový framework pro tvorbu webových API. Tento framework umožňuje vývojářům dosáhnout více s menším množstvím kódu a v kratším čase, což jej činí rychlým a efektivním nástrojem pro vývoj webových aplikací. V této práci byl společně s webovým serverem uvicorn [10] použit pro tvorbu backend části webové aplikace.

3.3.2 Bootstrap

Pro tvorbu frontend části byl využit framework Bootstrap [11], který je HTML5, CSS a JavaScript frameworkem s 72% [11] podílem na trhu. Používá se k rychlému a snadnému vytváření responzivních webových stránek. V době psaní práce je verze tohoto frameworku, Bootstrap 5.

3.3.3 Jinja2

Jinja2 [10] je populární šablonovací engine pro Python, který umožňuje vývojářům efektivně skládat HTML šablony. Ve spojení s FastAPI se Jinja2 používá k dynamickému generování HTML obsahu na základě backendové logiky. Tento přístup umožňuje oddělit prezentaci od aplikace, což zjednodušuje údržbu a rozšiřitelnost webových aplikací.

3.3.4 NGINX

NGINX [12] je webový server optimalizovaný pro rychlost a škálovatelnost. V této práci bude využit pouze jako webový server, který poskytuje symboly.

3.3.5 Symbol server

Jedním z míst kam můžeme symbolové a jiné soubory vznikající během kompilace, potřebné k ladění ukládat je symbol server. Jedná se o místo, které dokáže tyto soubory uložit a umožnit k nim později přístup.

Pro tento účel vznikl protokol SSQP[3] [13], pomocí kterého lze symboly, které je potřeba získat. Tento protokol specifikuje konvenci [14] pro dotazování na server. Protokol byl navržen tak aby doporučené konvence pro dotazování na soubory Windows byly kompatibilní s Windows SymSrv klienty.

[3]Simple Symbol Query Protocol

Symbol server tak usnadňuje proces ladění tím, že poskytuje přístup k informacím o programu, což umožňuje vývojářům analyzovat chyby, bez potřeby vývojáře dohledávat ručně soubory potřebné pro analýzu.

3.3.6 minidump-stackwalk

Jedná se o nástroj z kolekce knihoven rust-minidump, který je napsaný v programovacím jazyce Rust [15]. Tento nástroj, představující konzolové rozhraní poskytuje jak strojově čitelné, tak lidsky čitelné výstupy z minidumpu, včetně zpětných trasování [16] a symbolizace.

Použití tohoto nástroje bylo preferováno, neboť zpracování minidump souborů představuje složitý úkol. Dále je tento nástroj pravidelně udržován a jeho výstupy poskytují veškeré informace potřebné pro analýzu zaslaných minidump souborů, což odstraňuje nutnost vyvíjet nové řešení.

Tato kolekce Rust knihoven je dostupná na GitHub repozitáři [17]. Knihovna je licencovaná licencí MIT.

3.3.7 Docker

Docker [18] je platforma určená k vytváření, nasazování a provozu aplikací ve virtualizovaných kontejnerech, což zajišťuje snadnou přenositelnost a konzistenci mezi vývojovými a produkčními prostředími. Umožňuje zejména vytváření kompilačních kontejnerů, které existují pouze po dobu kompilace dalších kontejnerů. Tyto kontejnery jsou využívány přímo pro kompilaci a obsahují vše potřebné pro tento proces, zatímco pro finální běh aplikace již tyto data nejsou potřeba. Tato vlastnost bude využita v této práci.

4 Požadavky a návrh systému pro sběr událostí

4.1 Požadavky

Systém musí splňovat následující funkční požadavky:

- Přijímat události od vzdálených aplikací CrashReporter.
- Zpracovávat z událostí minidump soubory pro získání zásobníku volání a dalších relevantních informací.
- Analyzovat zpracované události a seskupovat je do kategorií podle společné příčiny.
- Rozlišovat mezi různými typy událostí.
- Poskytovat přístup k datům z analýzy prostřednictvím webové aplikace.
- Ukládat logovací (.log) a minidump (.dmp) soubory událostí pro možnost pozdější detailní analýzy vývojářem.
- Izolovat provoz jednotlivých aplikací pomocí Docker kontejnerů pro zvýšení bezpečnosti a nezávislosti komponent.

4.2 Návrh

Systém je navržen s ohledem na modularitu. Jednotlivé komponenty budou implementovány jako aplikace v samostatných Docker kontejnerech, které lze snadno spouštět a spravovat.

Celkový návrh systému je založen na postupném přesouvání událostí mezi moduly, které tyto události přijmou, zpracují, analyzují a následně umožní vývojářům přístup k výsledným analýzám.

4.3 Databáze

Vzhledem k různorodosti dat přicházejících do systému a jejich plánované transformaci je nezbytné vybrat technologii databázového serveru, která umožňuje měnit schéma uložené datové struktury.

SQL databáze by v daném případě musela obsahovat mnoho sloupců, což by zkomplikovalo proces přesunu jednotlivých dat z událostí. Proto se očekává výběr některé z NoSQL databází.

4.4 Modul pro příjem a zpracování záznamů událostí CrashReporter

Tento modul musí poskytovat koncový bod pro příjem těchto požadavků. Tyto požadavky je třeba zpracovat a tyto informace uložit do databáze.

Je důležité, aby odpověď na požadavek byla poslána co nejdříve. Kvůli případům, kdy zdroj události je makro Ensure a aplikace čeká na odpověď z koncového bodu a pro uživatele se chová jako kdyby „zamrzla“.

Informace z cesty v URL a zpracované tělo příchozího POST požadavku uložíme do kolekce určené pro úspěšně přijaté události. V případě chyby tyto požadavky uložíme do kolekce s dalšími informacemi, pro pozdější analýzu.

4.5 Modul pro zpracování přijatých událostí

Tento modul dále zpracovává soubory minidump přijatých událostí. U distribučních sestavení hry se běžně nedodávají symboly pro ladění koncovým uživatelům. Automaticky generovaný kontext CrashReporting systémem tedy nedokáže vytvořit informace potřebné pro pozdější analýzu příčiny. Nejsou tím pádem vždy spolehlivá nebo kompletní. Modul musí tedy provést analýzu minidump souboru (.dmp) a její výsledek zapsat k události pro pozdější analýzu.

V případě událostí přijatých z režimu běhu aplikace Editor se přeskakuje zpracování minidump souborů, jelikož u těchto typů událostí je předpoklad, že jsou dostupné symboly v době události, tedy kontext z CrashReporter aplikace obsahuje všechny potřebné informace pro analýzu. Případně je velká šance, že by analýza byla neúspěšná, neboť symboly nejsou dostupné a to v případech kdy si vývojáři sami kompiluji editor. Po zpracování je dokument události přesunut z příchozí kolekce do zpracované kolekce pro analýzu.

4.6 Symbol server

Pro zajištění funkčnosti modulu zpracovávajícího přijaté události, který k analýze minidumpů potřebuje symboly a další soubory z doby kompilace, je nezbytné mít dostupný alespoň jeden server tohoto typu. V této práci předpokládáme, že žádný takový server není dostupný a musí být implementován.

4.7 Modul pro analýzu zpracovaných událostí

Modul pro analýzu zpracovaných události, které dále seskupuje do „problémů“, což je skupina událostí se stejnou příčinou. Modul se musí rozhodnout z informací poskytnutých z kontextu a z informací získaných zpracováním minidump souboru o příčině události.

Příčina události je speciální struktura, která vzniká analýzou informací z kontextu a zpracovaného minidumpu. Je určená k tomu, aby se daly jednotlivé události porovnávat mezi s sebou a mohly se seskupovat.

Tato příčina se následně musí porovnat s již existujícími událostmi se stejnou příčinou. Pokud nalezne shodu, událost přiřadí do již existujícího problému. Při nenalezené shodě vytvoří problém nový.

V případě úspěšné analýzy přesune událost do kolekce pro zanalyzované události. Neúspěšně analyzované události končí v chybové kolekci pro pozdější analýzu.

4.8 Webová aplikace

Analyzované události je nutné zobrazit vývojářům v přehledné formě. Uživatelům jsou prezentovány jednotlivé problémy, přičemž při otevření detailů problému je zobrazena poslední událost patřící k danému problému. Pro vývojáře je zásadní, aby byly informace prezentovány jasně a strukturovaně, včetně následujících údajů:

- Typ události, který je pro lepší orientaci odlišen barvami
- Zásobník volání
- Chybová zpráva generovaná systémem CrashReporting
- Zpráva od uživatele

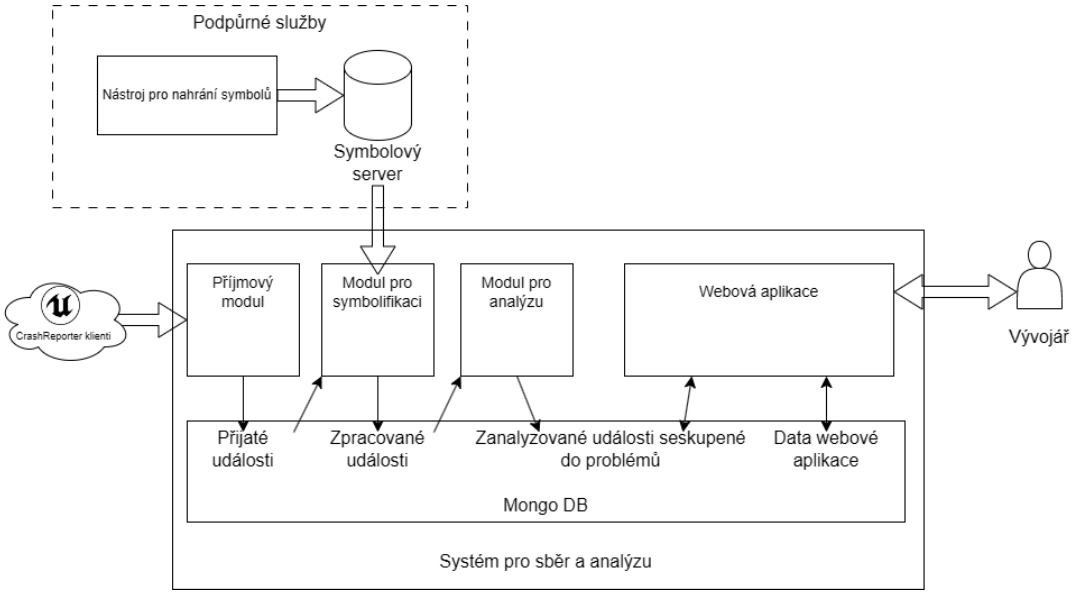
Dále je možné procházet jednotlivé události a sledovat jejich podrobnosti. V přehledu problémů je důležité zobrazit klíčové informace ke každému problému o:

- Celkovém počtu událostí daného problému
- Počtu událostí za časový úsek (24 hodin)
- Počtu individuálních uživatelů ovlivněných problémem

Použití barev pro odlišení typů událostí výrazně zlepšuje vizuální orientaci v datech a usnadňuje rychlé rozpoznání různých typů problémů, které systém identifikuje. Tato struktura a forma prezentace umožňuje vývojářům efektivně identifikovat problémy, což přispívá k rychlejšímu a efektivnějšímu vývoji aplikací.

5 Implementace

Systém je navržen tak, že jádrem systému jsou čtyři hlavní aplikace[4] (obrázek 2), které mezi s sebou vyměňují data pomocí databáze a ke kterým je podpůrnou službou symbolový server a nástroj, který umožňuje ukládat symboly na server.



Obrázek 2: Přehled implementace systému

5.1 Databáze

Pro ukládání dat byla v projektu zvolena databázová technologie MongoDB [19] z důvodu její schopnosti efektivně spravovat velké objemy dat a dynamicky se měnící struktury.

Flexibilita schématu MongoDB je zásadní pro adaptaci na změny ve struktuře dat, které vznikají díky různým typům zpracovaných událostí a také během zpracování událostí systémem.

Díky integrované komponentě GridFS byl také splněn požadavek na ukládání potenciálně velkých souborů (.log, .dmp) do jednotlivých záznamů událostí.

5.1.1 Struktura dat v databázi

Efektivní zpracování událostí, které systém přijímá, je zajištěno prostřednictvím využití více kolekcí v databázi MongoDB. Každý procesní krok, který musí být s událostí proveden, je reprezentován vlastní kolekcí (tabulka 7).

Události jsou tak postupně přesouvány mezi těmito kolekcemi v závislosti na jejich aktuálním stavu zpracování. Tento databázový model umožňuje efektivní

[4] Jejich složka, ve kterých lze aplikace najít v elektronických datech je v hranatých závorkách v jejich nadpisu

sledování stavu zpracování událostí, neboť jednoduchý dotaz na obsah specifické kolekce odhalí všechny události čekající na další zpracování.

Tento přístup je zvláště výhodný vzhledem k omezením MongoDB, které v rámci jedno serverového nastavení nepodporuje event-based dotazování na nově příchozí události.

Jméno kolekce	Účel
ingress	Přijaté události
processed	Zpracované události
issues	Problémy zanalyzovaných událostí
events	Zanalyzované události
web_settings	Nastavení webové části
ingress_errors	Chyby při příjmu událostí
analyze_errors	Chyby během analýzy

Tabulka 7: Kolekce v databázi

Každý modul při vytváření kolekce také zajišťuje vytvoření odpovídajících indexů, aby bylo zajištěno efektivní procházení a vyhledávání v datech.

Struktura dokumentu úspěšně přijaté události

Jméno pole	Datový typ	Význam
_id	ObjectId	MongoDB ID
datetime_received	Date	Datum a čas přijetí události
guid	String	GUID události
execution_guid	String	Execution GUID události
sender_ip	String	IP adresa odesílatele
request_path	String	Cesta v URL
crash_context	Object	Kontext převedený do objektu
compressed_minidump_id	ObjectId	ID souboru v GridFS
compressed_log_id	ObjectId	ID souboru v GridFS

Tabulka 8: Struktura úspěšně přijaté události

Struktura dokumentu události (tabulka 9) se nemění v průběhu zpracování, pouze se do ní přidávají pole. A to v případě úspěšného zpracování symbolizačním modulem. Přidává se pole „processed_dump“ typu Object, které zahrnuje data zpracovaného minidumpu. V případě neúspěšné symbolizace se pole nepřidává.

Jméno pole	Datový typ	Význam
_id	ObjectId	MongoDB ID
datetime_received	Date	Datum a čas přijetí události
guid	String	GUID události
execution_guid	String	Execution GUID události
sender_ip	String	IP adresa odesílatele
request_path	String	Cesta v URL
crash_context	Object	Kontext převedený do objektu
compressed_minidump_id	ObjectId	ID souboru v GridFS
compressed_log_id	ObjectId	ID souboru v GridFS
processed_dump	Object	Zpracovaný minidump

Tabulka 9: Struktura události po úspěšném zpracování

Další případem je když analyzující modul určí, pod jaký problém je událost zařazená, poté se přidává pole „parent_issue_id“ typu ObjectId, kde hodnota páruje dokument události s dokumentem problému a pole „crash_cause“ typu Object, což je objekt, který z dostupných dat vytvoří porovnatelný objekt (tabulka 10).

Jméno pole	Datový typ	Význam
_id	ObjectId	MongoDB ID
datetime_received	Date	Datum a čas přijetí události
guid	String	GUID události
execution_guid	String	EXECUTION GUID události
sender_ip	String	IP adresa odesílatele
request_path	String	Cesta v URL
crash_context	Object	Kontext převedený do objektu
compressed_minidump_id	ObjectId	ID souboru v GridFS
compressed_log_id	ObjectId	ID souboru v GridFS
processed_dump	Object	Zpracovaný minidump
crash_cause	Object	Objekt příčiny události
parent_issue_id	ObjectId	ID jako klíč přiřazeného problému

Tabulka 10: Struktura události po úspěšné analýze

Informace o problému, jako je čas prvního a posledního výskytu, událost, která ho vytvořila, a objekt příčiny pro seskupování událostí, jsou uchovávány v dokumentu problému (tabulka 11).

Jméno pole	Datový typ	Význam
_id	ObjectId	MongoDB ID
issue_create_datetime	Date	Datum a čas vytvoření problému
last_update_datetime	Date	Datum a čas poslední aktualizace
creating_event_datetime	Date	Datum a čas vytvoření události
creating_guid	String	GUID vytvářející události
creating_exec_guid	String	EXECUTION GUID vytvářející události
crash_cause	Object	Objekt příčiny události
summary	String	Shrnutí události

Tabulka 11: Struktura problému po analýze

5.2 Příjem dat od uživatelů [INGRESS_SERVER]

5.2.1 Architektura aplikace

Modul pro příjem dat je realizován jako Python aplikace běžící v Docker kontejneru. Pro svou funkcionalitu aplikace využívá následující externí knihovny dostupné přes pip:

- `pymongo` - Knihovna pro práci s MongoDB
- `xmltodict` - Knihovna pro převod xml souborů do Python slovníků

5.2.2 Implementace HTTP serveru

V rámci aplikace byl implementován HTTP server, který využívá integrované Python knihovny `http.server` a `socketserver`. Tento server byl implementován jako první modul systému, aby bylo zachyceno co nejvíce událostí pro následné zpracování. Je tedy primárně určen pro vývojové a testovací účely a jeho použití v produkčním prostředí není doporučeno. Implementuje pouze základní bezpečnostní kontroly, což může vést k bezpečnostním rizikům. V kapitole o budoucím vývoji systému 7 je plán tento modul přepsat aby využíval FastAPI jako technologii pro vytvoření HTTP koncového bodu.

Server je navržen pro vícevláknové zpracování požadavků, což umožňuje efektivně zvládat simultánní požadavky od různých klientů bez zbytečných zpoždění. Server naslouchá na portu 27274 uvnitř kontejneru, přičemž se předpokládá, že port bude zpřístupněn do sítě pomocí Dockeru.

Každý příchozí požadavek je zpracován třídou `crashRequestHandler`, která rozšiřuje třídu http.server.SimpleHTTPRequestHandler. Hlavní funkcionalita modulu je implementována v metodě, která zpracovává POST požadavky.

5.2.3 Funkcionalita

Po inicializaci aplikace dochází k načtení nezbytných konfiguračních dat z proměnných prostředí. Následuje připojení k databázi MongoDB s využitím knihovny

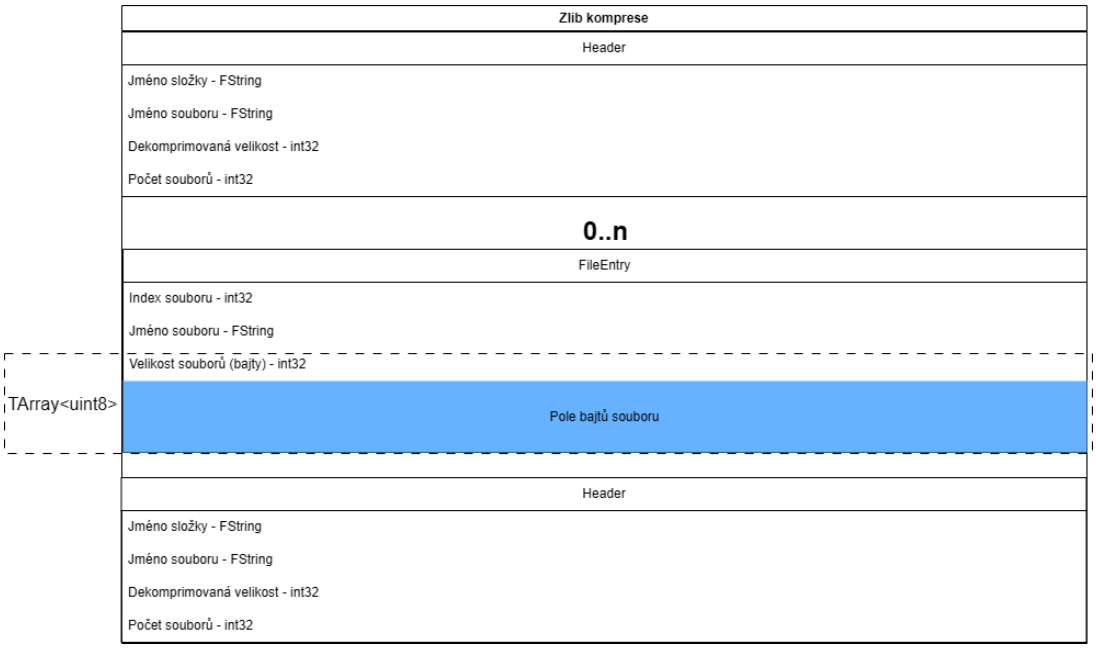
pymongo. Po úspěšném navázání spojení začne HTTP server naslouchat příchozím požadavkům.

Přijetí požadavku od CrashReporteru vyvolá proces dekomprese komprimovaných dat obsažených v těle požadavku (obrázek 3). Tato data jsou poté převedena do struktury umožňující efektivnější manipulaci s nimi, a to za použití knihovny specificky určené pro zpracování serializovaných dat z CrashReporteru popsaná níže (5.2.4). Tato knihovna poskytuje třídu `CrashReport`, která obsahuje funkce pro procházení jednotlivých souborů odeslaných na server. V případě, že mezi soubory je identifikován validní kontextový soubor CrashContext.runtime-xml, dochází k vytvoření datové struktury `události`.

Kontextový soubor je převeden do formátu slovníku pomocí knihovny `xmltodict` a následně uložen do proměnné `události`. Logový soubor a minidump jsou komprimovány pomocí `zlib` a uloženy do databáze za využití GridFS. Identifikátory těchto souborů jsou rovněž uloženy do proměnné `události`.

K informacím uloženým do proměnných `události` patří také IP adresa odesílatele, datum a čas přijetí požadavku a. URL cesta. Tyto informace jsou následně zapsány do kolekce určené pro přijaté `události`.

V případě, že dojde k chybě při zpracování, je chybové hlášení spolu s kompletním obsahem požadavku zapsáno do speciální kolekce v databázi pro účely pozdější analýzy.



Obrázek 3: Obsah těla požadavku poslaného pomocí CrashReporter

5.2.4 Knihovna crash_report_parser pro zpracování dat z CrashReportu

Pro zpracování dat z CrashReport struktury zaserializované pomocí Unreal Engine serializační metody byla vytvořena knihovna na deserializaci.
V zaserializovaných datech jsou dvě struktury:

Header je struktura, nesoucí v sobě informace o zaserializované složce jako je jméno složky (GUID události), jméno souboru [5], celková velikost souborů uložených ve struktuře a jejich počet (tabulka 12. Způsobem jakým CrashReporter tuto strukturu serializuje znamená, že data obsahují hlavičky dvě (obrázek 3). První má pouze vyplněné jméno složky a souboru. Druhá, která je až na konci, má vyplněné veškeré informace.

Datový typ	Informace
FString	Jméno složky
FString	Jméno
int32	Velikost dekomprimovaných souborů
int32	Počet uložená souborů

Tabulka 12: Zaserializovaná Header struktura

FileEntry je struktura, která v sobě nese informace nejen o souboru, ale obsahuje také jeho celý obsah v poli bajtů (tabulka 13).

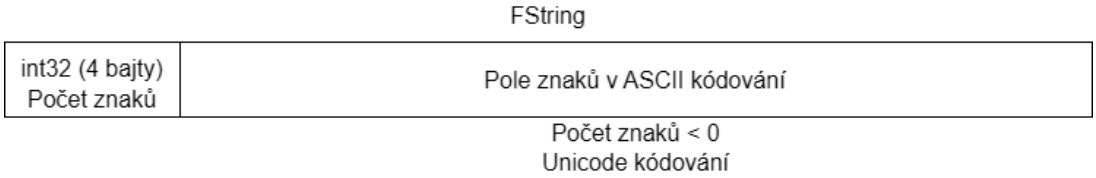
Datový typ	Informace
int32	Pořadí souboru ve struktuře
FString	Jméno souboru
TArray<uint8>	Pole bajtů souboru

Tabulka 13: Zaserializovaná FileEntry struktura

5.2.4.1 Deserializace struktury FString

Struktura FString funguje v této práci jako klasický řetězec. Na začátku této struktury se nachází 32 bitové číslo, označující počet znaků (včetně ukončující nuly), které se za číslem nachází (obrázek 4). Nemusíme v tomto případě řešit, zápornost počtu znaků, neboť CrashReporter posílá názvy v kódování ASCII, s minimální délku 260 znaků (Zbylé místo je vyplněné nulovými bajty)

[5] Jméno složky s příponou ".ue4crash"



Obrázek 4: Přehled FString struktury v paměti

5.2.4.2 Deserializace struktury TArray<uint8>

Jedná se o klasické pole bajtů. Před kterým je 32 bitové číslo označující počet bajtů které jsou v něm uloženy. Pro načtení dat tedy stačí zjistit počet bajtů přečtením tohoto čísla a následně přečíst tento počet bajtů.

Implementace deserializace v této práci neprovádí kontroly, které jsou dobrou praktikou v počítačové bezpečnosti. Jako jsou kontroly velikostí jednotlivých položek a jiné.

5.3 Zpracování přijatých dat [PROCESS]

5.3.1 Architektura aplikace

Modul na zpracování událostí je implementován jako Python aplikace běžící v Docker kontejneru. Ten potřebuje pro svůj běh nástroj rust-stackwalk, jehož sestavování probíhá ve speciálním dočasném kontejneru a následně je jeho výsledek překopírován do finálního kontejneru. To umožňuje oddělit kompilaci nástroje od samotné aplikace, která jej používá. Samotná aplikace, kromě nástroje také potřebuje pro svůj běh externí knihovnu pymongo pro práci s MongoDB.

5.3.2 Funkcionalita

Aplikace periodicky kontroluje dokumenty z příjmové kolekce, které jsou určené ke zpracování. Jednotlivé události, poté prochází. Pokud z kontextu události aplikace vyhodnotí, že je potřeba zpracování souboru minidump učiní tak. Z dokumentu události poté stáhne pomocí GridFS z databáze soubor minidump do dočasné složky. Aplikace vytvoří příkaz, kterým spustí nástroj rust-minidump a ve kterém specifikuje důležité spustitelné parametry pro správný běh nástroje. Jsou jimi:

- Cesta k minidump souboru ke zpracování
- URL adresa serveru se symboly
- Formát výstupu (JSON)

- Cesta ke složce pro uložení symbolů stažených nástroje rust-minidump
- Cesta ke složce sloužící jako mezipaměť během stahování a přípravy symbolů

Ke spuštění příkazu je použit Python modul subprocess, která jej vykoná a jeho standardní výstup zapíše do textové proměnné. Ze které je následně výstup načten pomocí modulu `json` do slovníku. Ten je následně přidán jako nové pole „processed_dump“ do dokumentu události. Tento dokument považujeme za zpracovaný. Může být tedy přesunut z příjmové kolekce do zpracované kolekce pro další krok.

V případě, kdy zpracování bylo neúspěšné nebo bylo přeskočeno se dokument přesune bez jakékoliv změny do zpracované kolekce pro další krok.

5.4 Analýza zpracovaných dat [ANALYZER]

5.4.1 Architektura aplikace

Modul na zpracování událostí je implementován jako Python aplikace běžící v Docker kontejneru. Pro svůj běh potřebuje pouze externí knihovnu `pymongo` pro práci s MongoDB.

5.4.2 Funkcionalita

Aplikace periodicky kontroluje dokumenty ze zpracované kolekce, které jsou určené k analýze. Dokument události, se poté načte knihovnou na třídu „Event“, obsahující informace z dokumentu zpracované pro další zpracování pro určení příčiny události a to třídou „Event_cause“.

Tato třída „Event_cause“ zpracovává informace z kontextu dodaného k události a snaží se vyhodnotit místo, ve kterém aplikace měla událost. K tomu využívá zásobník volání dodaný v kontextu a v případě úspěšném zpracování souboru minidump zásobník volání ze souboru. Tento zásobník volání je dále vyčištěn o rámce, které nejsou k analýze důležité.

Následně je tato třída připravena na použití pro vyhledávání v kolekci již existujících problémů, ve které se vyhledává primárně podle:

- Typu události
- Módu enginu v době události
- Jména souboru, kde událost nastala
- Čísla řádku v souboru, kde událost nastala s tolerancí (±3 řádky) [6]

[6] Tolerance vznikající rozdílným způsobem získávání rámců zásobníku volání. CrashReporter oproti nástroji rust-stackwalk

Tyto podobné události jsou dále porovnávaný na podobnost jejich zásobníků volání. Tato podobnost je implementována jako porovnání rámců události s případnými rámci existujících problémů.

Tento algoritmus, prochází rámce z obou příčin a vypočítá podobnost. Kvůli rozdílnosti výstupů jednotlivých metod získávání rámců zásobníků volání, se můžou jednotlivé jména funkcí lišit. Pro podobnost jmen tříd a funkcí v rámcích zásobníku volání byl proto použit upravený algoritmus na jejich podobnost založen na metrice Levenštejnova vzdálenost[7].

Při velké podobnosti rámců ve stejném pořadí je událost zařazena do problému. Do dokumentu události se přidají informace k jakému problému patří „parent_issue_id“ a dokument se přesune do kolekce analyzovaných událostí. Pokud událost nastala dřív, než první událost problému, aktualizuje se dokument problému.

Pokud neexistují žádné problémy, nebo podobnost není dostatečná, vytváří se nový problém do kterého se vloží informace o události, která jej vytváří, a dále příčina události struktura „Event_cause“. Tento problém je uložen do kolekce problémů a dokument události se přesune do kolekce analyzovaných událostí.

5.5 Webová aplikace [HUMAN_INTERFACE]

Backend webové aplikace je realizován jako Python aplikace využívající FastAPI, která je spuštěna v Docker kontejneru.
Pro svou funkcionalitu využívá následující externí knihovny dostupné z pip:

- `fastapi[all]` - FastAPI framework
- `motor` - Oficiální asynchroní ovladač pro MongoDB
- `uvicorn` - Asynchroní server, pro zpracování HTTP požadavků pro FastAPI

Frontend je vytvořen s využitím knihovny Bootstrap verze 5, která byla v době psaní práce nejaktuálnější. Využívá se předkompilované verze CSS a JS souborů, což eliminuje potřebu jakékoliv další kompilace.

Aplikace je pro přehlednost a udržovatelnost strukturována do čtyř FastAPI routerů[8], každý se specifickým účelem a sadou koncových bodů:

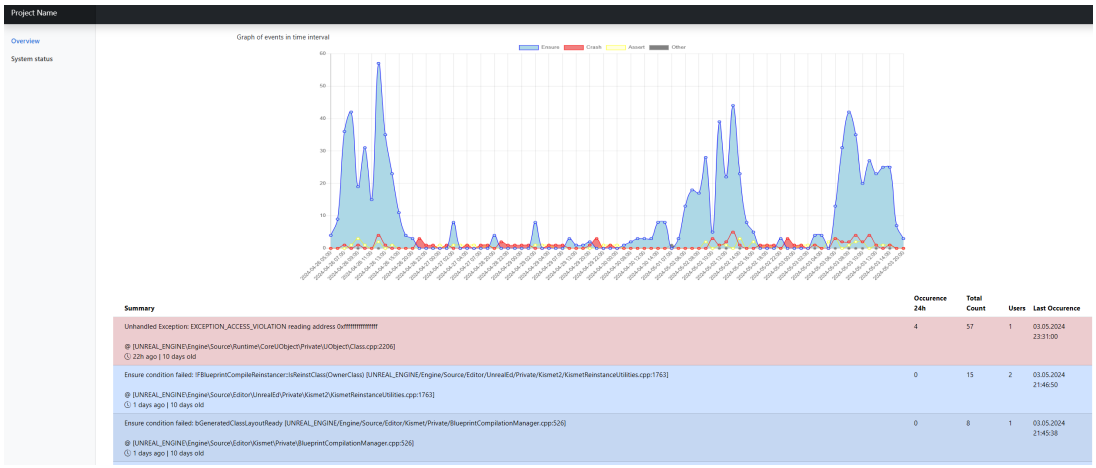
- **Router pro statické soubory**: Zajišťuje distribuci statických zdrojů, jako jsou obrázky, CSS a JavaScriptové soubory.
- **Router pro celkový přehled**: Poskytuje koncové body pro agregovaný seznam problémů v projektu.
- **Router pro detaily problémů**: Umožňuje přístup k detailním informacím o problémech a stahování jejich dat.

[7] Metrika pro měření editační vzdálenosti mezi dvěma řetězci
[8] Routery v FastAPI umožňují segmentaci API do logických jednotek

- **Router pro systémový přehled**: Nabízí koncové body pro monitorování celkového stavu systému a systémových zdrojů.

Pro generování HTML výstupu je využíván šablonovací engine Jinja2, který umožňuje vytvářet dynamické uživatelské rozhraní s použitím knihovny Bootstrap. Tato kombinace podporuje efektivní skládání komponentů uživatelského rozhraní díky modulární struktuře šablon Bootstrapu, což výrazně zjednodušuje a zefektivňuje vývoj frontendu.

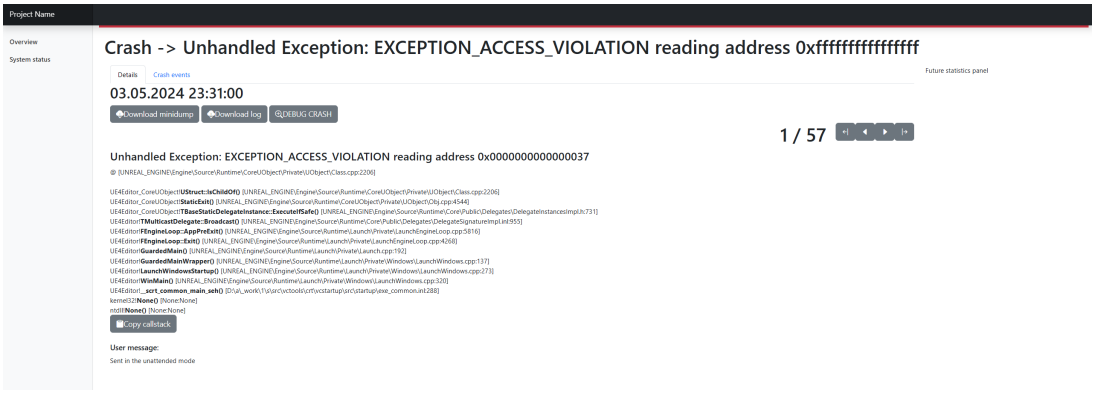
5.5.1 Přehledový pohled



Obrázek 5: Vzhled přehledové stránky s grafem

Aplikace zobrazuje přehledný výpis problémů (obrázek 5), které se v projektu vyskytly, včetně grafu počtu jednotlivých typů událostí za posledních 8 dní, agregovaných po 2 hodinách. Zobrazuje také datum prvního výskytu problému, datum poslední události, počet událostí přiřazených k problému za posledních 24 hodin a celkově, a počet unikátních zařízení, na kterých se události objevily.

5.5.2 Detailní pohled na problém



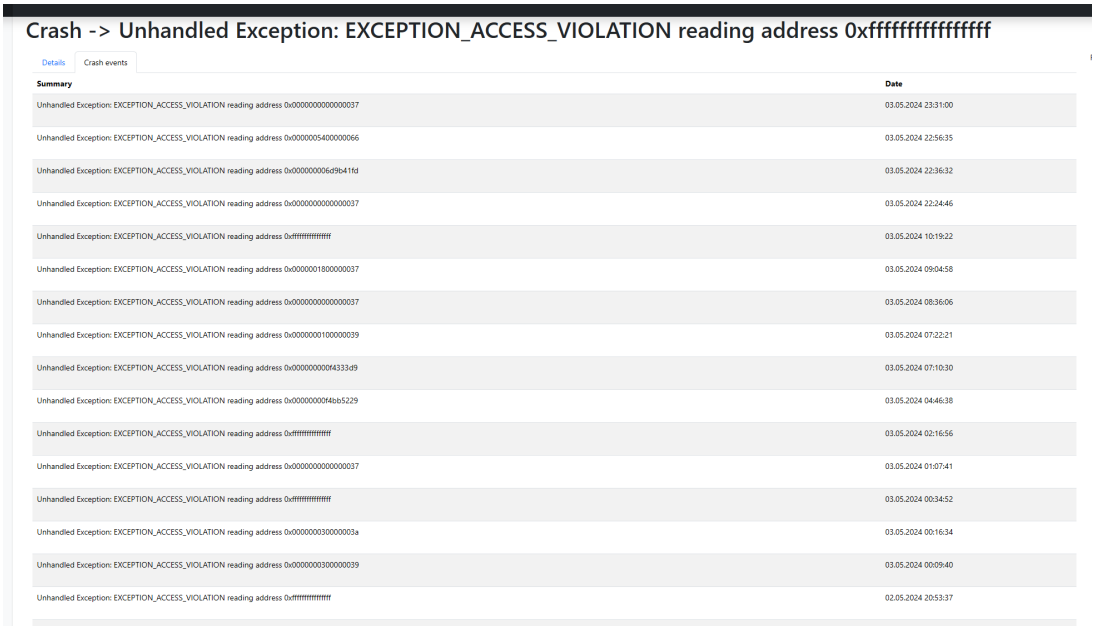
Obrázek 6: Vzhled detailního pohledu na problém

Detailní pohled na problém (obrázek 6) poskytuje informace o poslední přiřazené události, kterou systém zpracoval. Zobrazují se klíčové informace jako:

- Zásobník volání, který událost způsobil
- Typ události
- Chybová zpráva generovaná CrashReporting systémem
- Uživatelská zpráva

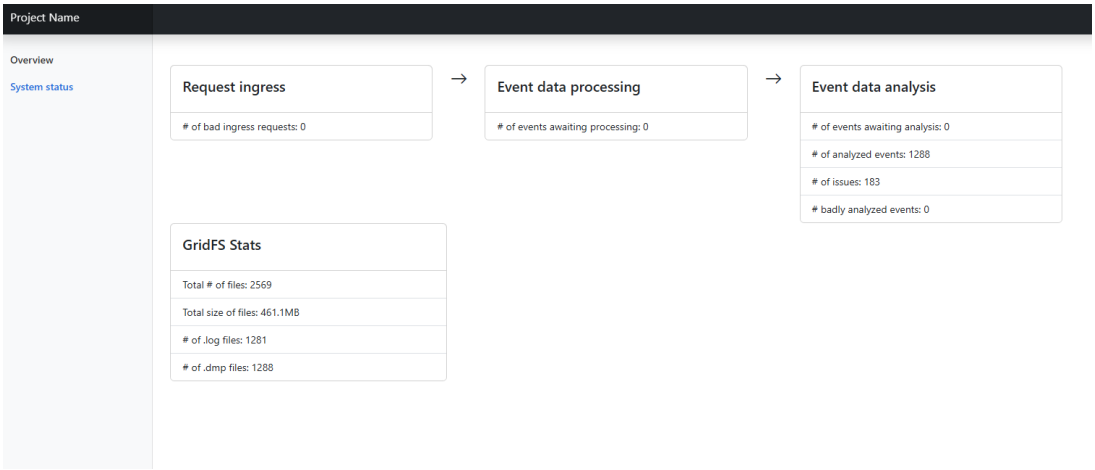
Události lze procházet od nejnovějších po nejstarší pomocí tlačítek na pravé straně. Zásobník volání lze pomocí tlačítka zkopírovat do schránky, pro rychlou analýzu vývojářem v jeho vývojovém prostředí. U každé události je možné stáhnout její minidump a logový soubor, v případě potřeby podrobnější analýzy.

Pro lepší představu o rozsahu a příčině problému, je také dostupný přehled jednotlivých událostí a chybových zpráv z CrashReporting systému (obrázek 7). Lze tak rychle a přehledně vidět, jestli dané události jsou způsobené například přístupem na nulový objekt (Uvidíme, že v chybové zprávě se přistupuje na adresy blízko nule)



Obrázek 7: Zprávy událostí přiřazených k problému

5.5.3 Stav systému



Obrázek 8: Přehled o stavu systému

Stav systému (obrázek 8) nabízí přehled jednotlivých kolekcí, které pracují s událostmi. U kolekcí, které využívá symbolizační modul nebo analytický modul, ukazuje počet událostí, které zbývají ke zpracování. Ukazuje také, kolik místa jednotlivé minidump a logové soubory zabírají. Najdeme zde také počet požadavků, které se nepodařilo přijmout nebo zanalyzovat.

5.6 Server pro hosting symbolů[SYMBOL SERVER]

Symbolový server ve zdrojových kódech v příloze je implementován jako jednoduchý kontejner obsahující server NGINX, který umožňuje pomocí HTTP přístup ke složce se symboly. Pomocí Dockeru je do této složky připojena další složka z hostitelského souborového systému, která je strukturována podle konvence SSQP pro pojmenování složek. K přípravě složky a jejímu naplnění soubory lze použít Python knihovnu `symstore` [20].

V reálném prostředí se předpokládá, že je zmíněný server již v rámci organizace existuje, nicméně řešení v této bakalářské práci může sloužit jako startovní bod k jeho založení.

Z výše uvedených důvodů je cesta k symbol serveru nastavitelná v rámci proměnné prostředí u symbolizačního modulu.

6 Překážky při vývoji

6.1 rust-minidump

V průběhu zkoumání použitelnosti nástroje minidump-stackwalk byla objevena experimentální grafická aplikace minidump-debugger, využívající stejnou knihovnu jako CLI nástroj minidump-stackwalk. Bylo ověřeno, že aplikace umožňuje zpracovat minidump soubor, včetně informací pro zpětné trasování a s funkční symbolizací.

Přestože grafická aplikace poskytovala správné výsledky, při použití CLI nástroje minidump-stackwalk byly zjištěny špatné výsledky způsobené neadekvátní symbolizací, což znemožňovalo získání informací pro zpětné trasování. Tato nesrovnalost mezi výsledky obou aplikací vedla k detailnímu zkoumání zdrojových kódů a ladění příčin problémů.

Po identifikaci a odladění problémů byly prováděny modifikace, které byly implementovány ve verzi V0.19.1 pomocí patche, jenž je přiložen ve zdrojových kódech práce. Pro verze novější než V0.19.1, které byly aktuální v době psaní práce, tento patch již nefunguje, jelikož došlo k zastarání a odstranění používaného způsobu získávání symbolů, který byl patchem opraven. Proto bylo na GitHubovém repozitáři projektu vytvořeno issue [21] s cílem zajistit, aby nástroj správně získával Windows symboly ze symbolového serveru.

6.2 Analýza příčin událostí a jejich seskupování

Při analýze jednotlivých příčin událostí byly identifikovány různé problémy spojené s analýzou dat. Tyto problémy zahrnovaly například neúplná data zasílaná některými klienty a velmi rozdílné zásobníky volání mezi kontextovými soubory a zpracovanými výstupy souborů minidump. V důsledku těchto faktorů nemusí řešení implementované v této práci dosahovat optimálních výsledků v těchto případech. Přesto byly výsledky získané v rámci testovacího nasazení považovány za dostatečně uspokojivé.

7 Budoucí vývoj systému

Pro zlepšení uživatelské zkušenosti je plánováno rozšíření systému o funkce pro přihlašování a notifikace prostřednictvím webhooků do externích služeb. Tato rozšíření umožní lepší integraci systému do každodenních pracovních postupů vývojářů. V budoucnu dojde také k výraznému zlepšení uživatelského rozhraní, což zjednoduší navigaci a zvýší přehlednost při práci s chybovými hlášeními.

Dále je v plánu implementovat vylepšené statistické nástroje, které poskytnou vývojářům hlubší porozumění pro identifikaci a řešení opakujících se problémů. Pro vývojáře systému bude zavedena analytika využití, umožňující lepší pochopení interakcí uživatelů se systémem.

Důraz bude kladen i na bezpečnost a udržitelnost systému, včetně zlepšení příjmového modulu pro možnost jeho vystavení na internet a automatického čištění zastaralých dat, což zajistí jeho robustnost a spolehlivost v prostředí s potenciálně vysokým objemem příchozích dat a hrozbami z internetu.

8 Testování

Během vývoje systému byla jako vstupní testovací data využita data z reálného prostředí projektu vyvíjeného v Unreal Engine. Systém zpracoval 22 tisíc reálných událostí od 68 vývojářů, z nichž bylo možné určit příčinu u 15919 a které byly seskupeny do 886 problémů.

Rozložení přijatých a zpracovaných událostí (tabulka 14) a problémů (tabulka 15).

Typ události	Počet událostí	Zastoupení
Ensure	13817	86,80 %
Assert	855	7,55 %
Crash	1202	5,37 %
Ostatní	45	0,28 %

Tabulka 14: Rozložení událostí podle typu

Typ problémů	Počet problémů	Zastoupení
Ensure	555	62,64 %
Assert	144	16,25 %
Crash	171	19,30 %
Ostatní	16	1,81 %

Tabulka 15: Rozložení problémů podle typu

Tato data ilustrují, proč je výhodné, že se makro Ensure provádí pouze jednou během života aplikace, protože nejvíce zpracovaných událostí systémem jsou události tohoto typu. Jak je vysvětleno v kapitole kontrolních maker pro makro Ensure 3.2.8.3.

Velký počet událostí, u kterých nebylo možné určit příčinu (přibližně 5 tisíc), lze připsat skutečnosti, že pro jejich diagnostiku chyběly potřebné symboly, které nebyly dostupné na vývojovém symbol serveru. Další část dat byla neúplná a zbytek posloužil jako testovací sada pro budoucí vylepšení systému.

Závěr

V rámci této bakalářské práce byl vyvinut a implementován systém pro sběr, analýzu a vizualizaci chybových hlášení u projektů běžících na Unreal Engine. Systém se skládá z několika klíčových komponent: modulu pro sběr chybových hlášení, modulu pro následné zpracování dat z chybových hlášení, modulu pro analýzu zpracovaných dat, webové aplikace pro jejich vizualizaci a serveru pro hosting symbolů. Tento přístup umožňuje budoucí zdokonalení jednotlivých částí systému, což přispívá ke zlepšení detekce a kategorizace chybových stavů a značně zjednodušuje následnou analýzu a opravy chyb ve finálním produktu.

Během vývoje bylo identifikováno a úspěšně vyřešeno několik problémů, které vznikly zejména kvůli chybnému chování použitého nástroje pro zpracování minidump souborů a chybným předpokladům při analýze příčin událostí. Tyto problémy podtrhují význam testování v raných fázích vývoje za účelem zajištění spolehlivosti nástroje v produkčním prostředí.

Výsledky testování naznačují, že systém poskytuje solidní základ pro efektivní zpracování a analýzu chybových hlášení. Přesto byly identifikovány příležitosti pro budoucí rozvoj a vylepšení systému, včetně rozšíření funkčnosti, zlepšení uživatelského rozhraní a integrace s externími nástroji a službami. Tyto vylepšení mají potenciál dále zvýšit užitečnost systému pro vývojáře.

V závěru lze říci, že bakalářská práce naplnila své cíle a předložila funkční řešení pro monitorování a analýzu chybných událostí v projektech na Unreal Engine. Vývoj tohoto systému přispívá k lepšímu porozumění a efektivnějšímu řešení různých typů chyb, což je klíčové pro zvýšení kvality a spolehlivosti finálních aplikací. Budoucí vývoj a implementace navržených vylepšení jistě povedou k ještě lepší integraci tohoto systému do vývojových procesů a pracovních postupů.

Z tohoto důvodu bude systém dále vyvíjen a poté nasazen do ostrého provozu v profesionálním herním studiu.

Conclusions

Within the scope of this bachelor's thesis, a system for the collection, analysis, and visualization of error reports in projects running on Unreal Engine was developed and implemented. The system consists of several key components: a module for collecting error reports, a module for subsequent data processing from error reports, a module for analyzing the processed data, a web application for their visualization, and a server for hosting symbols. This approach allows for future improvements of individual parts of the system, contributing to better detection and categorization of error states and significantly simplifying subsequent analysis and error correction in the final product.

During development, several problems were identified and successfully resolved, which arose mainly due to the incorrect behavior of the tool used for processing minidump files and incorrect assumptions in the analysis of the causes of events. These issues underscore the importance of testing early in development to ensure tool reliability in a production environment.

The test results indicate that the system provides a strong basis for the effective processing and analysis of error reports. Nevertheless, opportunities for future development and improvement of the system were identified, including extending functionality, improving the user interface, and integration with external tools and services. These improvements have the potential to further increase the system's utility for developers.

In conclusion, the bachelor's thesis has met its objective and presented a functional solution for monitoring and analyzing error events in Unreal Engine projects. The development of this system lead to a better understanding and more efficient resolution of various types of errors, which is important for enhancing the quality and reliability of final applications. Future development and implementation of the proposed enhancements will contribute to even better integration of this system into development process and workflows.

For this reason, the system will continue to be developed and then deployed in a professional gaming studio.

A Obsah elektronických dat

Na samotném konci textu práce je uveden stručný popis obsahu elektronických dat odevzdaných v systému katedry informatiky spolu s textem. Tato data jsou nedílnou součástí práce a tvoří (datovou) přílohu textu práce. Povinné položky struktury dat jsou:

`text/`
: Adresář s textem práce ve formátu PDF, a soubor .zip obsahující zdrojový kód teoretické části.

`README.md`
: Textový soubor s informacemi o způsobu použití systému.

`src/`
: Adresáře se zdrojovými kódy praktické části.

Literatura

[1] Games, Epic. *Crash Reporting in Unreal Engine* [online]. 2024 [cit. 2024-4-25]. Dostupný z: ⟨https://dev.epicgames.com/documentation/en-us/unreal-engine/crash-reporting-in-unreal-engine⟩.

[2] Games, Epic. *Crash Reporter Server - Crash Reporting in Unreal Engine* [online]. 2024 [cit. 2024-4-25]. Dostupný z: ⟨https://dev.epicgames.com/documentation/en-us/unreal-engine/crash-reporting-in-unreal-engine#crashreportserver⟩.

[3] Developer, Game. *Classic Tools Retrospective: Tim Sweeney on the first version of the Unreal Editor* [online]. 2024 [cit. 2024-4-29]. Dostupný z: ⟨https://www.gamedeveloper.com/design/classic-tools-retrospective-tim-sweeney-on-the-first-version-of-the-unreal-editor⟩.

[4] Games, Epic. *Unreal Engine End User License Agreement* [online]. 2024 [cit. 2024-4-25]. Dostupný z: ⟨https://www.unrealengine.com/en-US/eula/unreal⟩.

[5] GitHub. *Pull requests documentation* [online]. 2024 [cit. 2024-4-27]. Dostupný z: ⟨https://docs.github.com/en/pull-requests⟩.

[6] Chacon, Scott; Straub, Ben. *Pro Git* [online]. 2014 [cit. 2024-4-27]. Dostupný z: ⟨https://git-scm.com/book/en/v2⟩.

[7] Games, Epic. *Unreal Engine Programming and Scripting* [online]. 2024 [cit. 2024-4-27]. Dostupný z: ⟨https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-programming-and-scripting⟩.

[8] Games, Epic. *Asserts in Unreal Engine* [online]. 2024 [cit. 2024-4-29]. Dostupný z: ⟨https://dev.epicgames.com/documentation/en-us/unreal-engine/asserts-in-unreal-engine⟩.

[9] Bhat, Sathyajith. *Practical Docker with Python: Build, Release, and Distribute Your Python App with Docker*. 2022. ISBN 978-1-4842-7815-4.

[10] Lubanovic, Bill. *FastAPI*. 2023. Dostupný také z: ⟨https://www.oreilly.com/library/view/fastapi/9781098135492/⟩.

[11] Foreman, D.C.; Foreman, D. *Bootstrap 5 Foundations*. 2021. Dostupný také z: ⟨https://books.google.cz/books?id=KN1tzgEACAAJ⟩. ISBN 9798749992465.

[12] DeJonghe, Derek. *Complete NGINX Cookbook: Updated Edition*. 2022. Dostupný také z: ⟨https://www.nginx.com/resources/library/complete-nginx-cookbook/⟩.

[13] dotnet/symstore. *Simple Symbol Query Protocol* [online]. 2024 [cit. 2024-4-29]. Dostupný z: ⟨https://github.com/dotnet/symstore/blob/main/docs/specs/Simple_Symbol_Query_Protocol.md⟩.

[14] dotnet/symstore. *SSQP Key conventions* [online]. 2024 [cit. 2024-4-29]. Dostupný z: ⟨https://github.com/dotnet/symstore/blob/main/docs/specs/SSQP_Key_Conventions.md⟩.

[15] Klabnik, Steve; Nichols, Carol. *The Rust Programming Language* [online]. 2024 [cit. 2024-4-29]. Dostupný z: ⟨https://doc.rust-lang.org/book/title-page.html⟩.

[16] Ford, Ann R.; Teorey, Toby J. *Practical Debugging in C++*. Velká Británie: Prentice Hall, 2002.

[17] rust-minidump. *rust-minidump: A library for parsing minidump files*. 2024. Dostupný také z: ⟨https://github.com/rust-minidump/rust-minidump⟩.

[18] Mouat, Adrian. *Using Docker: Developing and Deploying Software with Containers*. Spojené státy americké: O'Reilly Media, 2015.

[19] Membrey, Peter; Hows, David. *MongoDB Basics*. 2014. ISBN 978-1-4842-0895-3.

[20] Foundation, Python Software. *symstore: Python Debug Symbol Server*. 2024. Dostupný také z: ⟨https://pypi.org/project/symstore/⟩.

[21] rust-minidump. *rust-minidump: Issue #970*. 2024. Dostupný také z: ⟨https://github.com/rust-minidump/rust-minidump/issues/970⟩.

[22] Cordone, Rachel. *Unreal Engine 4 Game Development Quick Start Guide: Programming Professional 3D Games with Unreal Engine 4*. Velká Británie: Packt Publishing, 2019.

[23] LogRocket. *How to debug Rust with VS Code* [online]. 2024 [cit. 2024-4-29]. Dostupný z: ⟨https://blog.logrocket.com/how-to-debug-rust-vs-code/⟩.