Laboratorio di Parallel Computing
Gennaro D'Imperio 7119465

Introduzione

L'obiettivo di questo progetto è di elaborare e categorizzare le voci dei log SSH per rilevare potenziali intrusioni di sicurezza. Le voci dei log sono state analizzate da un file di circa 420MB allo scopo di identificare varie categorie di attività sospette, utilizzando due approcci: elaborazione sequenziale ed elaborazione parallela in Python. Questa relazione fornisce una panoramica del progetto, descrive l'approccio di elaborazione parallela e discute i risultati delle prestazioni ottenuti con diversi approcci e diversi numeri di processi.

Metodologia

Ogni voce di log è stata categorizzata in base a specifici modelli e parole chiave per identificare potenziali problemi di sicurezza. Le categorie includono:

- Connessione chiusa [preauth]
- Utente non valido [preauth]
- Fallimento dell'autenticazione [preauth]
- Disconnessione [preauth]
- Mappatura inversa fallita [preauth]
- Mappatura sospetta
- Connessione chiusa
- Utente non valido
- Fallimento dell'autenticazione
- Disconnessione
- Attività sospetta
- Mappatura inversa fallita

La funzione di categorizzazione (categorize_log_entry) utilizza espressioni regolari e confronti di stringhe per assegnare ogni voce di log alla categoria appropriata.

Elaborazione Sequenziale

Nell'approccio di elaborazione sequenziale, il file di log è stato elaborato riga per riga. Per ogni voce, la categoria e l'indirizzo IP associato, sono stati determinati e memorizzati in un dizionario.

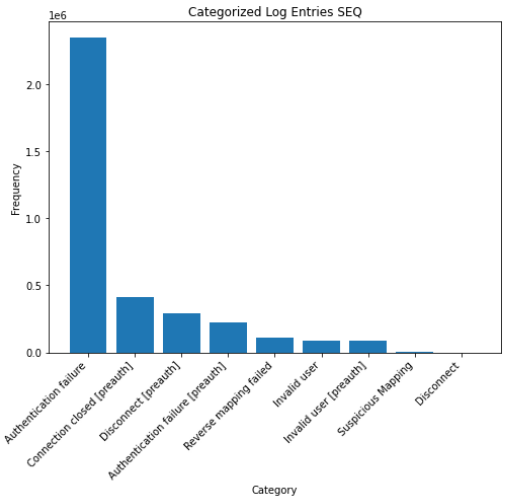
La frequenza di ciascuna categoria è stata poi calcolata e visualizzata utilizzando un grafico a barre. Inoltre vengono riportati gli indirizzi IP associati a più di una categoria nel file "ip.txt".

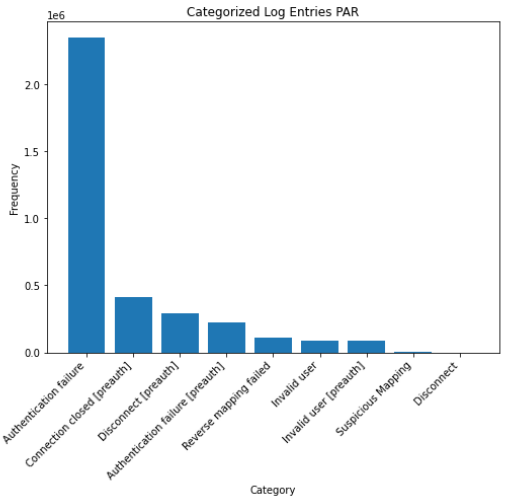
Elaborazione Parallela

L'elaborazione parallela è stata implementata suddividendo il file di log in blocchi e distribuendo questi blocchi su più processi. Ognuno di essi ha elaborato il proprio blocco di voci di log, categorizzando le voci e contando le occorrenze delle categorie. Anche se questa elaborazione è parallela, la scrittura degli indirizzi IP più frequenti nel file di testo "ip2.txt" avviene in modo sequenziale.
process_chunk(chunk): Elabora un blocco di voci di log, categorizzandole e contando le occorrenze delle categorie e degli indirizzi IP.
merge_results(results): Unisce i risultati parziali ottenuti dai vari processi in un unico set di dati aggregati.
process_log_parallel(log_file_path, num_processes): Gestisce l'intero flusso di lavoro di elaborazione parallela, suddividendo il file di log in blocchi, creando i processi e unendo i risultati finali.

Output

Il codice consente di visualizzare i plot delle categorie più frequenti riscontrate nei log, fornendo una rappresentazione visiva chiara e immediata dei risultati dell'analisi.
Grazie all'uso della libreria matplotlib, i dati raccolti e processati vengono presentati sotto forma di grafici a barre, permettendo un confronto diretto delle frequenze delle diverse categorie di eventi identificati. Oltre ai grafici, il codice salva su file i dettagli delle categorie di log per ciascun indirizzo IP che ha generato almeno due diverse categorie di eventi. Questo output fornisce un'ulteriore livello di dettaglio, utile per analisi più approfondite.
I file di output contengono righe nel formato "{ip}: {numero di categorie} categories: {categorie}", rendendo semplice la consultazione e l'interpretazione dei dati.





Prima Implementazione con 'multiprocessing.Manager'

Nel processo di sviluppo del progetto, una delle strategie iniziali per parallelizzare l'elaborazione dei file di log è stata quella di utilizzare "multiprocessing.Manager". Tuttavia, dopo una serie di prove ed analisi, è stato deciso di abbandonare questa strada in favore di un approccio differente. Di seguito vengono descritti i motivi di questa decisione. L'implementazione con Manager prevedeva l'utilizzo di strutture dati condivise tra i processi, come dizionari e liste

gestiti dal Manager. Questo approccio permetteva ai vari processi di aggiornare in parallelo un'unica struttura dati centralizzata. Nel contesto di questo progetto, ip_category_countsP era un dizionario gestito da un Manager per raccogliere e aggiornare i conteggi delle categorie di log associate a ciascun indirizzo IP.

Problemi Riscontrati con Manager

- **Overhead di Comunicazione**: L'utilizzo di un Manager comportava un overhead significativo per la comunicazione tra i processi. Ogni aggiornamento alla struttura dati centralizzata richiedeva una sincronizzazione, che ha rallentato considerevolmente il processo, specialmente nel caso di molti aggiornamenti concorrenti.
- **Scalabilità Limitata**: Questo approccio non scalava bene con un numero crescente di processi. L'overhead di gestione delle risorse condivise diventava un collo di bottiglia, riducendo l'efficienza complessiva del parallelismo.
- **Complessità del Codice**: L'uso di Manager richiede una gestione attenta della sincronizzazione, aumentando la complessità del codice e il rischio di errori. Questo si verifica spesso quando si aggiornano frequentemente strutture dati condivise.

```
77
78  def process_log_parallel(log_file_path, num_processes):
79      start_time = time.time()
80      with open(log_file_path, "r") as log_file:
81          lines = log_file.readlines()
82      chunk_size = len(lines) // num_processes
83      chunks = [lines[i:i + chunk_size] for i in range(0, len(lines), chunk_size)]
84
85      with Pool(processes=num_processes) as pool:
86          results = pool.map(process_chunk, chunks)
87
88          for ip_counts, intrusion in results:
89              for ip, categories in ip_counts.items():
90                  if ip is not None:
91                      ip_dict = ip_category_countsP.get(ip, {})
92                      for category in categories:
93                          ip_dict[category] = ip_dict.get(category, 0) + 1
94                      ip_category_countsP[ip] = ip_dict
95      end_time = time.time()
96      execution_timeManager = end_time - start_time
97      print("Manager execution time: {:.2f} seconds".format(execution_timeManager))
...     ....
120 if __name__ == '__main__':
121
122     manager = Manager()
123     ip_category_countsP = Manager().dict()
```

```
In [22]: runfile('C:/Users/Empir/.spyder-py3/parallel_corrected.py', wdir='C:/Users/Empir/.spyder-py3')
Sequential execution time: 24.99 seconds
Manager execution time: 20.95 seconds
Parallel execution time: 22.07 seconds with 8 processors
Speedup: 1.13
```

Seconda Implementazione con 'collections.defaultdict'

Per affrontare le problematiche sopracitate, si è deciso di utilizzare "collections.defaultdict", una struttura dati fornita dalla libreria standard di Python che offre numerosi vantaggi:

- **Semplicità del Codice**: "defaultdict" semplifica la gestione delle chiavi mancanti in un dizionario, creando automaticamente una nuova voce con un valore predefinito quando si accede a una chiave inesistente. Questo riduce la necessità di controlli espliciti per la presenza delle chiavi, rendendo il codice più pulito e leggibile.

- **Prestazioni Migliorate**: Utilizzando "defaultdict" in combinazione con la suddivisione del lavoro in chunk locali, ogni processo può lavorare su una copia locale dei dati. Questo elimina l'overhead di sincronizzazione tra processi, migliorando significativamente le prestazioni. Al termine dell'elaborazione parallela, i risultati parziali vengono uniti in una fase successiva. Questo approccio "divide et impera" consente di sfruttare al meglio le risorse di calcolo parallelo, minimizzando i punti di contenimento.

Impatto sulle Prestazioni

L'adozione di defaultdict ha avuto un impatto positivo significativo sulle prestazioni del sistema. I test comparativi tra l'approccio originale basato su Manager e il nuovo approccio con defaultdict hanno mostrato una significativa riduzione dei tempi di esecuzione.

Conclusioni Finali

Dopo aver riscritto l'implementazione parallela del codice, sono stati effettuati alcuni test con numeri di processi diversi. È evidente un miglioramento significativo nei tempi di esecuzione rispetto all'elaborazione sequenziale e quella precedente, in particolare con 4 e 8 processi.

- **4 Processori**: Qui lo speedup è molto positivo. Con quattro processori, il carico di lavoro è ben bilanciato e l'overhead è relativamente basso, portando a un notevole miglioramento delle prestazioni.

```
In [8]: runfile('C:/Users/Empir/.spyder-py3/parallel_corrected.py', wdir='C:/Users/Empir/.spyder-py3')
Sequential execution time: 25.31 seconds
Parallel execution time: 15.68 seconds with 4 processors
Speedup: 1.61
```

- **8 Processori**: Anche in questo caso, lo speedup è molto positivo. L'elaborazione parallela continua a beneficiare di una maggiore suddivisione del lavoro, sebbene l'overhead inizi a diventare più significativo.

```
In [18]: runfile('C:/Users/Empir/.spyder-py3/parallel_corrected.py', wdir='C:/Users/Empir/.spyder-py3')
Sequential execution time: 26.76 seconds
Parallel execution time: 15.33 seconds with 8 processors
Speedup: 1.75
```

  Tuttavia, l'aumento del numero di processori oltre 8 ha portato a un aumento dei tempi di esecuzione, probabilmente a causa dell'overhead di gestione dei processi e della comunicazione tra essi.

- **2 Processori**: La suddivisione del lavoro tra due processori riduce il tempo di esecuzione, il miglioramento è presente ma non ottimale.

```
In [14]: runfile('C:/Users/Empir/.spyder-py3/parallel_corrected.py', wdir='C:/Users/Empir/.spyder-py3')
Sequential execution time: 27.80 seconds
Parallel execution time: 19.72 seconds with 2 processors
Speedup: 1.41
```

- **16 Processori**: Con sedici processori, il tempo di esecuzione aumenta rispetto a 8 processori. L'overhead di gestione dei processi e della comunicazione tra di essi appiattisce quasi i benefici della parallelizzazione, portando a una riduzione dell'efficienza complessiva.

```
In [5]: runfile('C:/Users/Empir/.spyder-py3/parallel_corrected.py', wdir='C:/Users/Empir/.spyder-py3')
Sequential execution time: 25.38 seconds
Parallel execution time: 21.80 seconds with 16 processors
Speedup: 1.16
```

Aggiornamento 05/07:

Parallelizzazione del Blocco di Codice

In seguito alla revisione del laboratorio in data 05/07/2024, è stata implementata la parallelizzazione del blocco di codice responsabile della scrittura degli indirizzi IP con categorie multiple nel file di output. L'obiettivo era migliorare ulteriormente le prestazioni sfruttando il parallelismo tramite la libreria 'concurrent.futures' che è ottimizzata per la gestione del threading in Python.
L'adozione della parallelizzazione tramite la libreria sopra citata, ha contribuito a migliorare leggermente lo speedup complessivo del processo per diversi motivi:

- Evitare che un singolo thread monopolizzi l'accesso al file, permettendo a più thread di eseguire operazioni di scrittura simultaneamente. Ciò ha ridotto il tempo complessivo dedicato a questa attività.
- Sovrapposizione delle operazioni riducendo i tempi di attesa e migliorando il throughput del sistema.

```python
def write_ip_txt(ip, categories):
    with open(ip_path2, "a") as output_file:
        if len(categories) >= 2 and ip:
            output_file.write(f"{ip}: {len(categories)} categories: {categories}\n")

def process_log_parallel(log_file_path, num_processes):
    with open(log_file_path, "r") as log_file:
        lines = log_file.readlines()

    chunk_size = (len(lines) + num_processes - 1) // num_processes
    chunks = [lines[i:i + chunk_size] for i in range(0, len(lines), chunk_size)]

    with Pool(processes=num_processes) as pool:
        results = pool.map(process_chunk, chunks)

    ip_category_countsP, intrusion_signs = merge_results(results)

    with ThreadPoolExecutor(max_workers=num_processes) as executor:
        futures = [executor.submit(write_ip_txt, ip, categories) for ip, categories in ip_category_countsP.items()]

    for future in futures:
        future.result()

    category_counts = defaultdict(int)
    for category in intrusion_signs:
        category_counts[category] += 1
```

Parallelizzare anche la fase di scrittura ha permesso di sfruttare meglio le capacità multitasking del sistema, riducendo i tempi di esecuzione complessivi e migliorando lo speedup del blocco parallelo anche se di poco.