עבודת הגשה מספר 1

קורס: נושאים מתקדמים בפרטיות ואבטחת מידע (67515)

מגיש: דור מסיקה, ת.ז: 318391877

12 בדצמבר 2022

קובץ עיצוב

ארכיטקטורה

הסבר הארכיטקטורה שבניתי:
$Server$ - מכיל אובייקט של עץ בינארי מושלם, ומספר משתנים נוספים לפשטות הגישה אליו. בנוסף, מכיל פונקציות GET שונות על מנת שה־$Client$ יוכל לקבל ממנו מידע, ובפרט ללא בקשה לקבלת כל המידע השמור על הסרברת שיפגע בדרישות הזיכרון.
$Client$ - מבחינת זיכרון, שומר בתוכו, מילון שמפתחותיו הם id (הוגדר כי קיים סרבר בודד לקליינט) וכל ערך מערך המכיל את המסלול עבור אותו עלה המשוייך ל־id וערך בו נשתמש בהמשך עבור אימות הדאטה ($Authentication$). בנוסף, יכיל גם מפתח הצפנה פרטי שלו, על מנת להצפין את כל הערכים שיכניס לסרבר, ואיתו יפענח אותם בעת גישה אליהם.
מלבד מימוש הפונקציות הנתונות ב־API של התרגיל, בסוף ביצוע אחסון ערך חדש בסרבר (שכפי שהגדרנו, יתבצע תמיד בשורש), נבצע דחיפה כלפי מטה בעץ למניעת $Overflow$, כפי שראינו בהרצאה:
בכל איטרציה עבור מספר הרמות בעץ יבחרו שני קודקודים שונים באופן אקראי באותה רמה (פרט לראשונה, בה יש רק קודקוד אחד ולכן הוא יבחר לבד), ובבאקט של כל קודקוד יבחרו באופן אקראי שני איברים, אשר אלו ידחפו רמת אחת כלפי מטה בעץ על פי המסלול המתאים שלהם לעלה אליו הם משויכים. אם יבחר $Dummy$, הוא ידחף באופן שרירותי לאחד הבנים של הצומת בו הוא נמצא. כל איבר שנדחף יוחלף ב־$Dummy$ על מנת לשמור על שלמות כל באקט (למטרות $security$).
◄ גם לאחר קבלת $data$ מסוים מהסרבר, תופעל מתודת אחסון של האיבר שהתקבל חזרה שורש העץ, יחד עם שיוך לעלה חדש באותו סרבר.
◄ בעת איתחול של סרבר חדש, המשתמש ימלא אותו בערכי $Dummies$, לשם שמירה על הבטיחות כפי שמוסבר במאמר, לאחר שהוצפנו גם כן.
◄ לכל מידע המאוחסן בסרבר הוספנו ביט ראשון המבדיל בין $Dummy$ למידע של הקליינט, כאשר אם ביט זה הוא 0 (לאחר הפענוח כמובן) נוכל לדעת שמדובר ב־$Dummy$.

איך היא מספקת את הדרישות:
- אם קיים סרבר אחד עבור משתמש אזי זכרון המשתמש הינו אכן $O(N \cdot log\ N)$ - מינימלי כנדרש.
- בכך שבכל שמירת או לקיחת מידע מהסרבר אנו שומרים אותו מחדש בשורש ומשנים את המסלול המתאים עבורו, נקבל את האבטחה המתבקשת לכך שה־$Client$ יסתיר את תבנית הגישה למידע הסרבר המרוחק, כפי שמוגדר $ORAM$. אכן מתקיים כי גישה כפולה לאותו קובץ הינה גישה לשני מסלולים רנדומליים ובלתי תלויים אחד בשני.
- באופן בו ממומשות כל פונקציות הגישה לסרבר, נקבל כי כל פעולה מתבצעת בזמן לגוריתמי (לצד ביצוע פעולת הדחיפה), שכן לכל היותר עוברת על מספר השכבות בעץ, כמתבקש.
- כל מידע חדש שנכנס לסרבר מוצפן מראש על ידי המשתמש, ובנוסף בעת כל אחסון מידע (המתבצע תמיד בשורש) תתבצע הצפנה מחדש של כל איברי הבאקט של שורש העץ, כך שכל הקבצים יהיו בלתי ניתנים להבחנה.
- בעת קבלת מידע מהסרבר, יבדק כי תקין על פי הפורמט אותו הוא אמור לקבל, ובנוסף יוודא כי ה־$data$ שלו לא השתנה מזה שאוחסן תחילה ($data\ integrity$), בהתאם ל־id ול־$data$, זאת על מנת למנוע קבלת מידע שהשתנה בזדון על ידי גורם חיצוני הפועל על הסרבר.

ביצועים

$Throughput\ (number\ of\ requests\ per\ second)\ vs\ N\,(DB\ Size)$

על מנת לקבל את התוצאות שמוצגות בשני הגרפים הבאים, נדגם גודל בסיס נתונים N שעבורו בוצע מספר מקסימלי של פעולות לאורך 10 שניות.

גודל בסיס הנתונים נלקח בכל איטרציה החל מ־2^3 ועד 2^{12}.

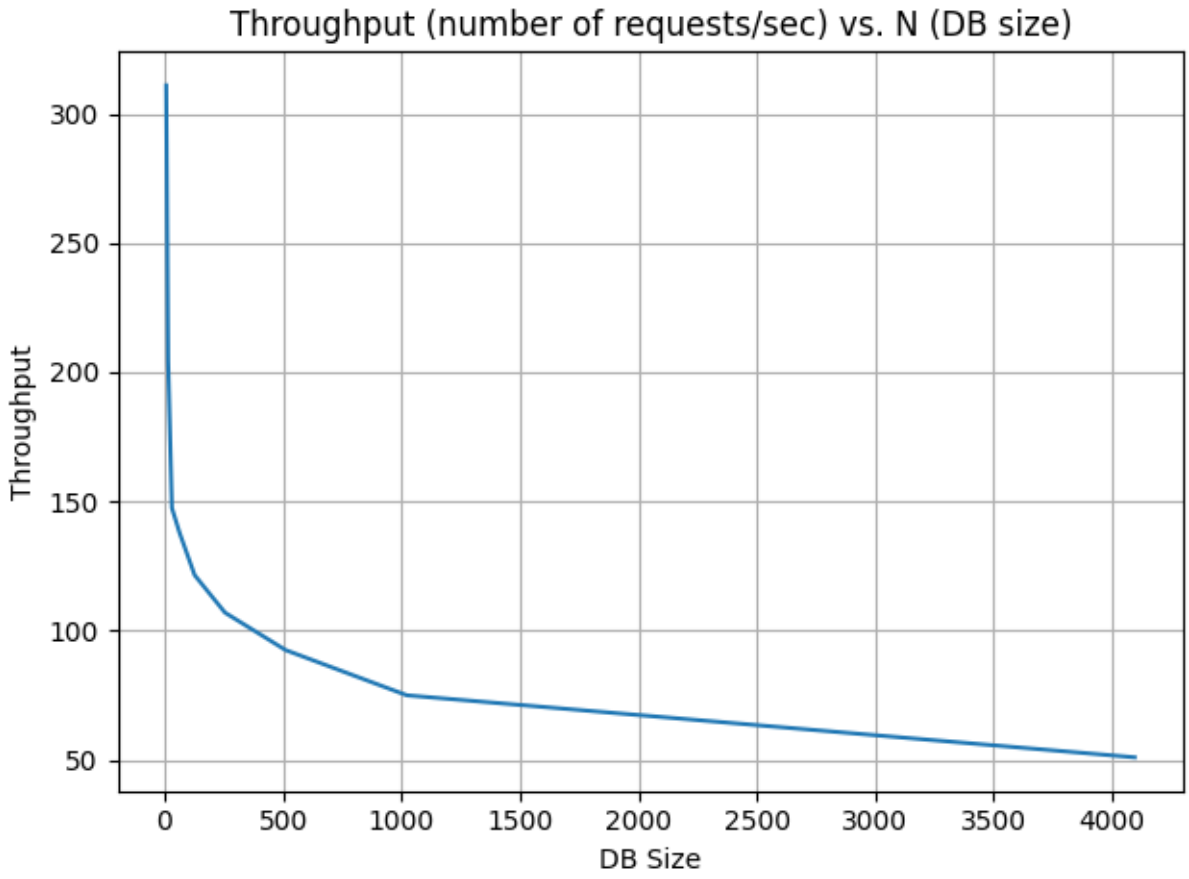
עבור כל איטרציה ה־$Throuput$ חושב על ידי מספר הפעולות שבוצעו חלקי מספר השניות שעבר, כלומר $\frac{number\ of\ requests}{10}$, וה־$Latency$ חושב על ידי זמן הביצוע של הפעולות חלקי מספר הפעולות שבוצעו, ובעצם $\frac{10}{number\ of\ requests}$.

<u>יחידות המדידה בהן השתמשתי:</u>

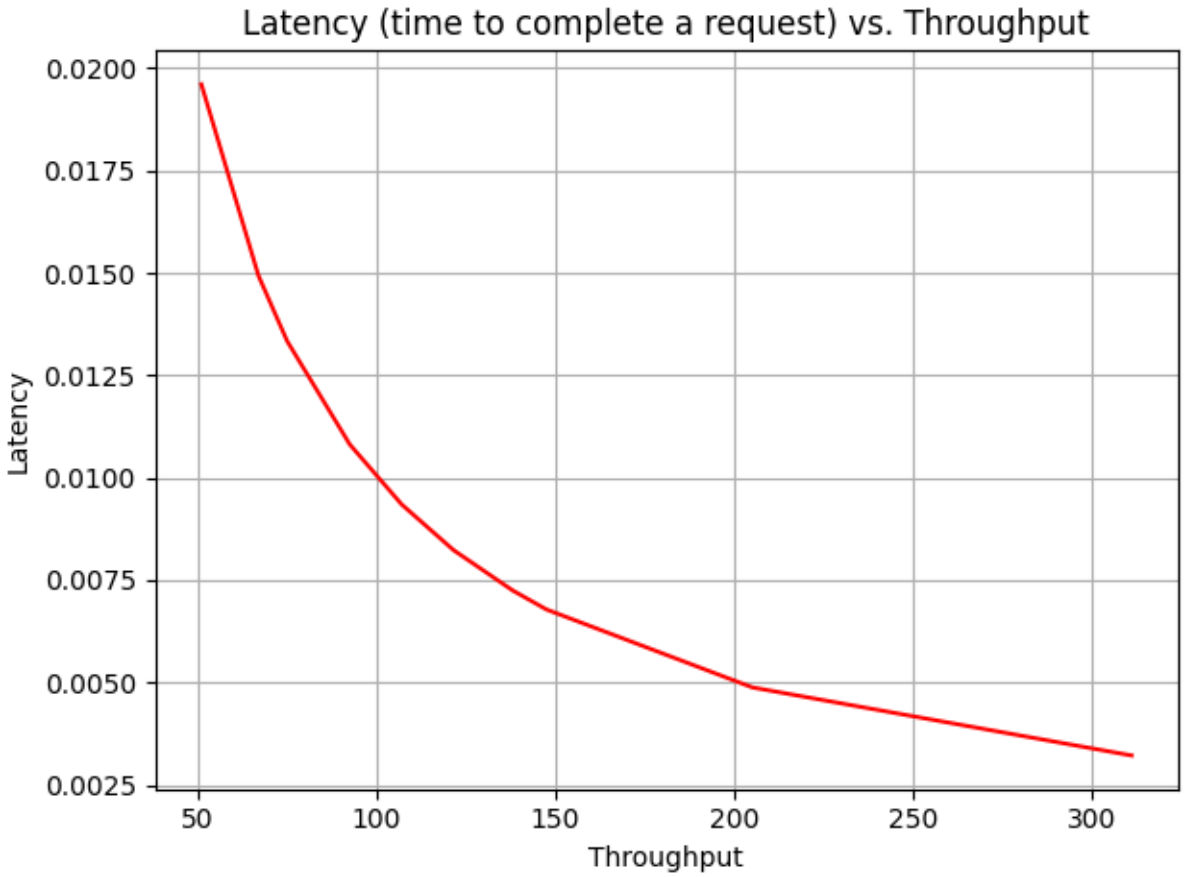
$Throughput$ - בקשות לשנייה.

בסיס הנתונים - מספר הצמתים בעץ, כשגודל כל אחד קבוע.

$Latency$ (עבור הגרף השני) - מספר השניות שלקח ל־$Client$ לבצע פעולה.



$Latency\ (time\ to\ complete\ a\ request)\ vs\ Throughput$



האם המימוש מרוויח משימוש ב־$multicore$?

כן. אם נדאג שלא יכנסו שני תהליכים לאותו קודקוד יחד (על מנת למנוע גישה כפולה למילון שיכולה לגרום לבעיות בזיכרון הסרבר), נוכל להרוויח מכך שיפור של יעילות האלגוריתם, בעיקר בעת פעולת דחיפת הערכים מטה בעץ. יעילות זו תבוא לידי ביטוי בכך שנוכל בעצם לבצע את כל הפעולות השונות המוגדרות ב־API באופן מקבילי ולא טורי, ובכך לשפר באופן ישיר את ה־$Latency$ וה־$Throuput$.